UNIVERSITY OF OSLO
Department of Informatics

Model-Level Back-in-Time Debugging of State Machine Systems

Master thesis
60 credits

Jonas Winje

May 4, 2009

Abstract

When a program failure occurs, the cause of that failure cannot always be found in the reached state of the program. Back-in-time debuggers address this issue by storing information about the program's execution history, but face challenges where performance and scalability are concerned.

Our approach is restricted to systems of state machines. The transitions can execute at nearly full speed since we merely save data from one state machine after each transition. Our debugger offers a model-level view of the running system, dealing with transitions and signal passing rather than individual code statements.

We offer empirical data on time and space overheads and we have evaluated the usability of our debugger on a set of students in a course with UML modelling. And we discuss the problems associated with reverting the state of a system during runtime.

Acknowledgements

I would like to thank my supervisor, Øystein Haugen, for good ideas, advice and feedback. And I would like to thank all the students who have tested the debugger, made suggestions and given advice. Thanks to Bjørn Brændshøi for the *HighlightPapyrusDiagramElement* routine, and to Rayner Vintervoll for making JFT work in increasingly stable versions of Papyrus.

Oslo, Norway. May 4, 2009

Jonas Winje

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 Back-in-Time

With a traditional debugger, only the reached state of the program under debugging can be inspected. A back-in-time debugger, on the other hand, stores information about the execution history of the program under debugging. This makes it a lot easier to locate the causes of some program failures. As noted in [1], a defect in a running program can cause an "infected state" long before that infected state causes a failure that an observer will see. Finding the "root cause" of a program failure can be difficult. In bug isolation experiments run by Liblit et al. in [2], they found that the call stack trace contained essentially no information about the cause of a bug when some symptom of it occurred.

1.1.2 Model-Level

We are dealing with state machine systems that have been modelled and then transformed to code. The development of the system happens on model level, so we want to present model-level information about it to the developers.

Omniscient debugging (section 7.1) stores the the entire execution trace of the program under debugging. In [3], Pothier et al. notes that one of the challenges of omniscient debugging is that the user must navigate potentially huge event traces. With the debugger working on model-level, rather than code-level, we reduce the number of stored events (the size of the trace), as each event spans over several code statements.

1.2 Contributions and Goals

The main contributions made in this thesis are:

- An approach to, and implementation of, back-in-time debugging of state machine systems, combining state machine-local checkpoints with logging of events that have effects outside one state machine. Our debugger lets us navigate the execution history of the system we are debugging and view the state of that system. And it lets us revert the state of the system we are debugging to what it was at the point in time we have moved to.

- Evaluation of the usability of the debugger, based on students using it in an UML modelling course.

- Overhead measurements of the debugger being used on test systems.

We also discuss problems that can arise when we want to revert the state of a system to what it was at an earlier point in time and then resume execution of the system from that state. And we discuss some possible improvements of the approach and debugger, addressing known issues.

Our goals are that this approach to back-in-time debugging offers functionality that developers find useful, and that it is a viable approach for many different state machine systems.

1.3 Method and Thesis Structure

In chapter 2 we give some background information on models and modelling languages, and describe the toolchain we use for developing the state machine systems our debugger is used on. We present our approach to back-in-time debugging in chapter 3, and the design and implementation of the debugger in chapter 4.

The usability evaluation is presented in chapter 5. The evaluation gives us an indication of whether or not the debugger functionality is useful to developers, and of how we can improve the debugger to make it more useful.

Scalability and overhead tests are presented in chapter 6. We measure the overhead in time and space caused by our debugger for different test systems. In addition, we determine the platform requirements of our approach (sections 2.4, 3.1 and 8.1). Together, the tests and the platform requirement should give us an indication of how viable our approach is.

We compare our approach to other approaches to back-in-time debugging in chapter 7. In chapter 8 we discuss and address known issues, in particular problems related to resuming execution from earlier points in time, and we discuss some possible improvements of our approach. Finally, conclusions and ideas for future work are presented in chapter 9.

The contents of the included CD are explained in appendix A. User instructions for the debugger are in appendix B.

Chapter 2

Background

In this chapter we will introduce the different things that make up the domain of this thesis and the background for the following chapters. We will give a brief overview of modelling and model transformation, and of the toolchain we use to create state machine systems.

2.1 Models and Metamodels

The state machine systems we are dealing with are modelled in UML and transformed to Java code. In the chapters following this one we will assume some familiarity with modelling languages, UML 2.x in particular, and model transformations.

Software development which focuses on modelling software at high abstraction levels using problem space domain concepts (rather than computational domain concepts), is known as Model-driven Engineering (MDE) [4]. The Object Management Group's (OMG) take on MDE is called Model-Driven Architecture (MDA) [5,6]. When using MDA, the idea is to first create a Computationally Independent Model (CIM), and then use automated model transformations, along with some manual adjustment, to decrease the abstraction level.

The way we create Java code from UML models is not strictly the MDA way to do it (e.g. we only do one transformation, from model to code), but it is related to it (and MDA is the best known MDE initiative there is).

2.1.1 Metalayers

The metamodel of a model is a model that specifies the modelling language used. The elements used in a model are instances of elements found in its metamodel. For example, if we create a new type of state machine in a UML model, that will be an instance of the element *StateMachine* found in the UML metamodel. The metamodel and the model are two "metalayers".

Different languages have different numbers of metalayers. For UML we have four: user object (e.g. an instance of the type of state machine we created in the model; M3), user model (M2), metamodel (M1) and meta-metamodel (M0). The meta-metamodel of UML (the metamodel of UML's metamodel) is the Meta-Object Facility (MOF) [7]. MOF is the meta-metamodel used for several OMG languages, and it is its own metamodel. For a metamodel to be MDA compliant, MOF must be the M0 metalayer.

2.1.2 UML 2.x

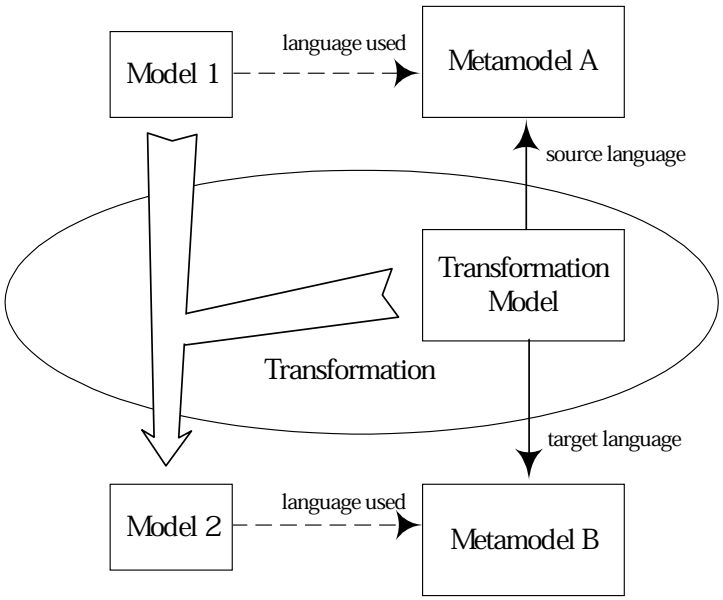
The Unified Modeling (UML) Language [8,9] is a general-purpose modelling language. The debugger we have created, is used on systems that have been transformed from UML 2.x models. For this reason, even though other modelling languages (such as SDL) can be used to create the kind of systems where our approach would be viable, UML is the most important one for us.

UML defines thirteen diagram types that are divided into the three categories: structure diagrams, behaviour diagrams and interaction diagrams (a subset of behaviour diagrams).

For our systems, we are interested in a few of the diagram types. Class diagrams (structure) to create classes and signals used by the state machine system. Composite structure diagrams (structure), to connect state machines to each other, and to the "edge" of the system, in order to allow for signals being sent between state machines and out of/from outside of the system. And state machine diagrams (behaviour) are used to specify the state machine behaviour for each of the state machines. Activity diagrams (behaviour) may be used, to specify the behaviour of state machine transitions and operations, but are not strictly necessary (and for our debugger, it makes no difference if code is transformed from an activity diagram of not[1]). Sequence diagrams (interaction) are typically used for describing interactions in the system, but are not part of what is transformed to Java code. (In section 2.3.2 we will look further into how the diagram types are used when using JavaFrame Transformation; for more details on how to model this type of state machine system, see [10].)

Modelling tools can be used to create diagrams and models. There are a lot of different UML tools available. There are lightweight tools that can be used to quickly sketch diagrams, with little or no support for an actual model. And there are other more heavyweight tools that can be used to create large models, where each model can have many diagrams, each diagram showing a different partial view of the model. As we are transforming from models to Java code, a tool that can be used to create models, and not just diagrams, is a necessity.

[1] To be precise, no code is transformed from any *diagrams*; for our debugger it makes no difference if code is transformed from the UML metamodel element *Activity* or something else.



(figure from [6])

Figure 2.1: The model transformation can be used on any model that uses A as its metamodel

2.1.3 Model Transformation

In MDA, a model-to-model transformations can be used to create a Platform Independent Model (PIM) from a CIM, and then a Platform Specific Model (PSM) from the PIM. Model-to-code transformation is used to generate code from the PSM. OMG has specified Query/View/Transformation (QVT) for doing model transformations. QVT only addresses model-to-model transformations and not model-to-code (or code/text-to-model) transformations. Model transformations are defined on metamodel level, i.e. one metalayer above the models that they can be used on. One model transformation can then be used for any model using the metamodel the transformation uses (figure 2.1). For example, to transform a UML model, the transformation must use the UML metamodel. For developing our state machine systems we use model transformation, but only to transform from models to code. The models we use are comparable to a PSM in MDA: we use UML models (platform independent, but not computationally independent) that use a platform specific UML profile.

2.2 Eclipse

Eclipse [11] is an open development platform. It has an extendible architecture, and through different plug-ins it supports development in many

languages.

The Eclipse Modelling Framework (EMF) project [12] offers support for metamodelling and model-to-model transformations. It includes the meta-metamodel Ecore, which is a variation of MOF. Other Eclipse projects and plug-ins in the model-driven domain typically use Ecore as meta-metamodel. The Modeling Tools Development (MDT) project [13] offers metamodel implementations and modelling tools for modelling languages. We use Papyrus [14], which is part of MDT, to model our state machine systems. Also part of MDT is UML2, an implementation of the UML 2.x metamodel. UML2 uses Ecore (rather than MOF) for its meta-metamodel, and Papyrus uses the UML2 metamodel.

We do not do any model-to-model transformations, but just transform from UML models to Java code. JavaFrame Transformation (section 2.3.2) to transform from UML model to Java code. JavaFrame Transformation makes use of Java Emitter Templates (JET), which is part of Eclipse's Model To Text (M2T) project [15].

UMLet. UMLet [16] is a lightweight UML diagram editor for Eclipse (available both in an Eclipse plug-in and in a stand-alone version). It can only be used to create diagrams, and has no support for models; its purpose is to provide a diagram editor that is fast and easy to use. With our debugger, the recorded history of execution can be exported to UMLet sequence diagrams.

2.3 JavaFrame and JFT

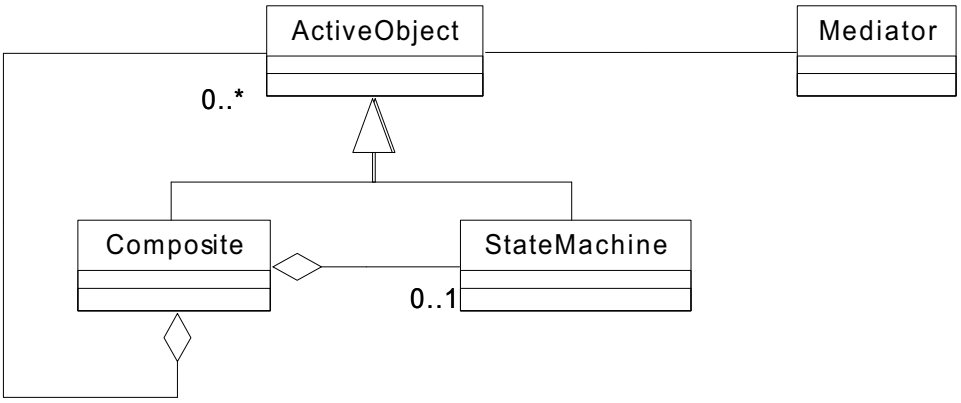
JavaFrame [17] is a Java framework for Java enabled modelling. JavaFrame is used to create state machine systems. JavaFrame Transformation (JFT) is an Eclipse plug-in that compiles (or transforms) UML models to Java. The code generated by JFT uses JavaFrame.

JavaFrame and JFT are used by student in projects in the courses *INF2120 - Project in Modeling*[2] and *INF5150 - Unassailable IT-systems*[3], and the tool developed for this thesis is used together with JavaFrame and JFT.

2.3.1 JavaFrame

A JavaFrame system is a system of state machines, composites and mediators (figure 2.2). A JavaFrame system has one outermost composite, and composites can contain other composites and state machines. Parts within a composite can be connected to each other, and to their enclosing composite,

[2] `http://www.uio.no/studier/emner/matnat/ifi/INF2120/index-eng.xml`
[3] `http://www.uio.no/studier/emner/matnat/ifi/INF5150/index-eng.xml`



(figure from [17])

Figure 2.2: JavaFrame concepts

through mediators. State machines can send and receive signals through their associated mediators.

States. Each state machine is in a state, and has a set of available states in a structure of composite states and states. The state that a state machine is in is found in its `state` variable. Composite states can contain states and other composite states, and each state must be within a composite state. Each such set of states has one outermost composite state, and all state machines of the same type share the same set of available states.

In addition to the `state` variable, a state machine can have any number of member variables, declared in a subclass of `StateMachine`. Some of these are mediators, and are not modified during a state machine's life cycle. All other member variables (including `state`) are part of what we will call the state machine's *full state* (all its data that can be modified during a regular transition).

Transitions. The state machines interact with each other and with other systems by sending and receiving signals. When state machines receive signals they execute transitions. Between transitions the state machines are inactive and their states and member variables will not change. During a transition a state machine's state can change, values of its member variables can change, and it can send new signals.

With our approach, the atomic step in time is one transition: our debugger can only pause the executing system *between* transition, and when it is paused all the points in time we can navigate to are between transitions.

Mediators. All signals are sent through mediators, and a state machine can send and receive signals through its associated mediators. Each of medi-

ator of a state machine can be used for either only outgoing signals, or only incoming signals. When a signal is sent, it is sent from a state machine, or from a mediator belonging to the outermost composite (an edge mediator), and is forwarded through mediators until it reaches a state machine or an edge mediator.

A state machine is connected to the rest of the JavaFrame system only through its mediators. When a state machine is created its mediators are set up. And when it terminates, all pointers to its mediators are removed from all other mediators. We do not consider a state machine's mediators part of its full state: mediator connections only change when a state machine is created or terminated, and changes typically happen in mediators that belong to other state machines than the one the current transition is for (e.g. if one state machine terminates during a transition, then pointers to that state machine's mediators are removed from *other* state machines' mediators).

Edge mediators are used for input to and output from the state machine system. For example, in the INF2120 and INF5150 projects, edge mediators are used to send and receive SMSes to and from cell phones. One `SMSInputMediator` that sends signals when the system is receiving SMSes, and one `SMSOutputMediator` that sends SMSes to cell phones when it receives certain signals.

2.3.2 JavaFrame Transformation

Versions of the JavaFrame Profile and JavaFrame Transformation are available for both IBM Rational Software Modeler (RSM) 6 and Papyrus UML. This section describes how state machine systems can be modelled with UML and compiled to JavaFrame systems, using a JavaFrame profile and JFT. We are using Papyrus, and because Papyrus and RSM 6 use different versions of the UML2 metamodel, some details will be different when modelling with RSM 6.

JFT Mapping. JavaFrame Transformation performs a mapping between UML elements and Java/JavaFrame elements:

- UML classes using the Composite stereotype from the JavaFrame profile become subclasses of JavaFrame's `Composite`.
- Regular UML classes become regular Java classes.
- UML state machines become subclasses of JavaFrame's `StateMachine`.
- UML regions, composite states and sub-state machines all become subclasses of JavaFrame's `CompositeState`.
- Regular UML states become subclasses of JavaFrame's `State`.

- UML signals become subclasses of JavaFrame's `Message`.
- UML Classes using one of the router mediator stereotypes become subclasses of the JavaFrame class corresponding to that stereotype (which again is a subclass of the `Mediator` class).
- All UML ports become mediator type member variables of the state machines and composites they belong to.
  - UML ports with a type become member variables of that type. That is, the UML class used as the port's type will be mapped to a Java class, and that Java class is used as the type of the member variable.
  - UML ports using the multicast mediator stereotype become `MulticastMediator` type member variables.
  - UML ports with no type and no stereotype become `Mediator` type member variables.
- UML properties for classes, state machines and signals become member variables of the Java classes their owners are transformed to.

In order to model a JavaFrame system, we must model a structure of composite classes and state machines, and then connect these to each other with ports and connectors.

Diagrams. Class diagrams are used to create regular classes, composite classes and signals.

For every composite class there should be a composite structure diagram. Composite classes can contain properties of state machine and composite class types. These are parts in the composite class's composite structure diagram. In the diagram, ports are added to the composite class and its parts, and the ports are connected to each other with connectors. The connectors are directional and define where the ports/mediators will send incoming signals.

State machine diagrams are made for the state machines. The regions, different kinds of states and transitions of state machines are created in state machine diagrams. The signals sent by state machines and used as triggers for transitions are created in class diagrams.

Activity diagrams may also be used, though these are not necessary to make a working JavaFrame system. Whenever a modeller can specify the effect of a transition with an activity diagram, ha can choose to use an opaque action instead. Opaque actions only contain Java code.

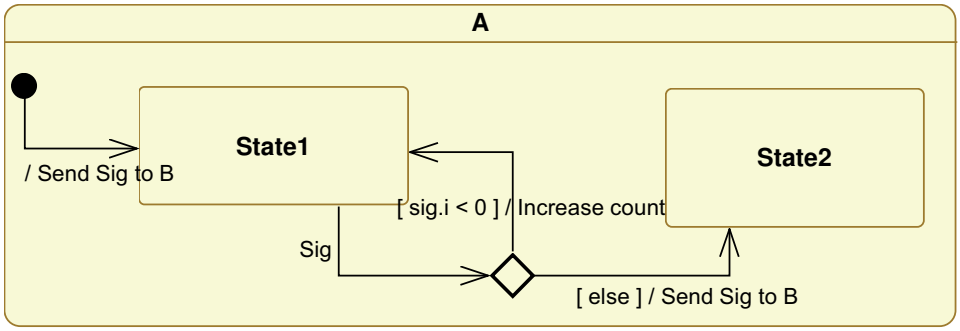


Figure 2.3: State machine diagram with Java code

Java code. State machine diagrams and activity diagrams will usually contain some Java code that is copied and pasted during the JFT transformation. The transition from one state to another is handled by JFT, but in order for the transition to have other effects, like sending a signal or changing the value of one of the state machine's member variables, some Java code must be added. E.g. in figure 2.3, the *Send Sig to B* effects and the *Increase count* effects are opaque actions containing Java code.

JFT will translate control structures (choice/junction pseudo states in state machine diagrams, and decision/merge nodes in activity diagrams) to if/else constructs, but Java code must be added to the guards of outgoing transitions from choice points, and control flows from decision nodes. In figure 2.3, `sig.i < 0` and `else` is Java code.

The modeller does not need to know how JavaFrame or JFT works in order to model systems, but he must know how to do a few JavaFrame specific things with the code he is adding. For example, he must know that the state machine's member variables can be accessed through a local variable `csm` (for "current state machine"), that the signal that triggered the current transition is available in a local variable `sig`, and he must know how to use the `output` method to send new signals through ports.

JFT Compared to an "Ideal" UML Compiler

JFT compiles UML models to Java code that uses JavaFrame. Because JavaFrame does not support all modelling concepts supported by UML, some information in the UML model is lost in translation when compiling to Java code. In addition, some UML concepts are not as well supported by JFT as others.

Transitions. JavaFrame does not have equivalents of the different kinds of UML pseudo states. The UML final state is transformed to a regular JavaFrame state named "FinalState", and the initial state of a JavaFrame state machine is `null`. Other pseudo states are never reflected in a state

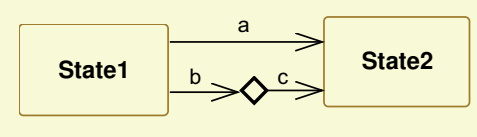


Figure 2.4: One JavaFrame transition can consist of both the *b* and *c* UML transitions.

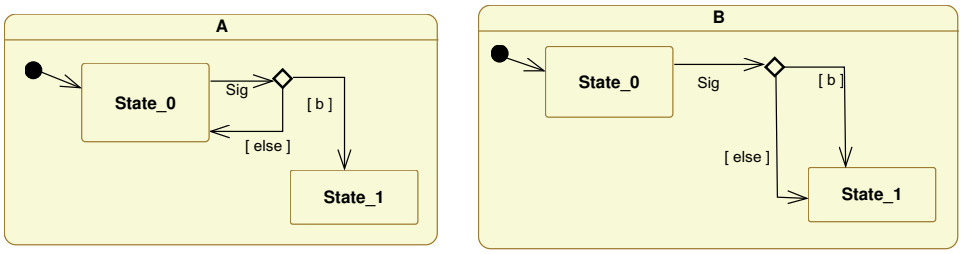


Figure 2.5: We cannot know which one is the correct transition path from *State_0* to *State_1* in state machine *B*.

machine's `state` variable. Transitions to and from pseudo states such as the choice and junction states are merged when transforming from UML transitions to JavaFrame transition. A JavaFrame transition consists of everything caused by a received signal, while in the UML model a signal can cause any number of transitions to and from pseudo states before reaching a regular state. In figure 2.4, the *a* transition would be translated to one JavaFrame transition, while both the *b* and *c* transitions would become part of the same JavaFrame transition.

In effect, from JavaFrame's point of view we cannot always tell what UML transitions just occurred after executing a transition, which is something we would like our debugger to be able to do. Knowing what signal caused a transition, and the state before and after the transition, it is often possible to find the UML transitions the JavaFrame transition consisted of. A path of UML transitions can be constructed from the before state, starting with the transition triggered by the signal, to the after state. However, if several such transition paths are found, it is impossible to tell which one is the right one. Figure 2.5 shows an example. Given a JavaFrame transition triggered by *Sig*, from *State_0* to *State_1*, for state machine *A* it would be possible to know what two UML transitions had occurred, as only one matching path of transitions exist. For state machine *B* it would not be possible without more knowledge about what went on within the JavaFrame transition.

In addition, because choice and junction states are transformed to if/else constructs, JFT does not handle cycles in such transition paths, such as a transition from one choice point going to the same choice point. Control paths through decision and merge nodes in activity diagrams have the same

issue. So if the modeller needs to create a loop of some kind, it must either involve a regular state (more than one signal processed, and more than one JavaFrame transition), or all of it must be written as Java code.

Inner classes. A UML class containing another class is not transformed to a Java class with an inner class. The same goes for all other UML elements that are transformed to Java classes. A state machine containing another state machine will not be transformed to a subclass of `StateMachines` with inner class that is a subclass of `StateMachine`, and so on. Instead of becoming an inner class, it will become a class in the same package as its owner.

For submachine states, this is an issue. Typically, we would like a state machine to be defined within another and use it as a submachine state in its owner's state machine diagram. And then, from the inner state machine, we would expect to have access to the member variables of both the inner and the outer state machine. The way this is handled by JFT, only the inner state machine's *behaviour* is used (i.e. its UML region or JavaFrame composite state), and only member variables declared in the outer state machine can be accessed from from the submachine. However this is only a (potential) issue for the modeller of a system. It does not create any problems for our debugger.

2.4 State Machine Systems

In this thesis we propose an approach to back-in-time debugging. The debugger we have created, JFDebug, relies on UML modelling with Papyrus or IBM RSM 6, model transformation with JFT, and JavaFrame. However we believe that this approach can be viable for other state machine systems as well. In this sectin we will define what we mean by state machine systems, as well as the components such systems must consist of. We can consider these some initial requirements for potential systems: systems meeting these requirements are the kind of systems we're interested in (even though our approach is unlikely to be viable to *all* such systems; the requirements will be discussed further in sections 3.1 and 8.1).

- By **state machine system** we mean a set of state machines that interact by **signal** passing. Signals can be sent between state machines in the system, from outside the system to state machines in the system, and they can be sent out of the system from state machines inside it.
- A **state machine framework**, then, is a framework offering support for state machine systems.
- A **state machine** is an entity (process, object, ...) with a full state.

- The **full state** is made up of **state** and **properties**.
  We use "state" to mean the state of a state machine, e.g. an instance of the UML element *State*. In an object-oriented context, "state" would usually mean *all* the data of an object. To avoid confusion, we use "full state" when that is what we mean (or all the data that can be mutated during a regular transition; in JavaFrame's case, that means all the data of the object, except its mediators.
- A state machine is an instance of a **state machine type**. The type determines what properties the state machine has and what different states and transitions are possible.
- Similarly, a signal is an instance of a **signal type**. The signal type determines what properties the signal has.
- It is only during **transitions** the state machine can modify its full state and send signals.
- All received signals trigger transitions.
- For a given state machine, incoming signals trigger transitions in the order they are received.
- The signal received and the full state of the state machine receiving the signal determines the behaviour of the transition executed.

Chapter 3

Approach

In this chapter we will outline our approach to back-in-time debugging. We will present the general design for such debuggers without getting into implementation details or features of particular state machine systems and frameworks.

In short, our approach is to copy the full states of state machines between their transitions, and to record important events during transitions. For this, we rely on three things:

- Clearly defined transitions. We know when a transition starts and when it stops.
- When a state machine is executing a transition, that transition can only modify the full state of that one state machine.
- We are able to discover important events as they occur during transitions.

We need to know when a transition start and when it stops in order to intercept execution of the system at those points. Figure 3.1 shows a sequence diagram for a series of transitions. The transitions are triggered by receiving signals, and one signal is sent during each transitions. The numbered, horizontal lines show where we want to create checkpoints (before and after each transition). After a state machine has executed a transition, we know that only that state machine's full state has been modified. At checkpoint 2 we only need to look for changes in a's full state, at checkpoint 3 we only need to look at b, etc., assuming that we know the state of the entire system at checkpoint *1*. And if we know the full state of a at checkpoint *1*, the full state of b at *2*, and the full state of c at *3*, then we know the state of the entire system at checkpoint *1*. Because we know that b's state is the same at checkpoint *1* and *2*, etc.

Although only one state machine's full state can be modified during one transition, certain events (such as signals being sent) do have effects on the

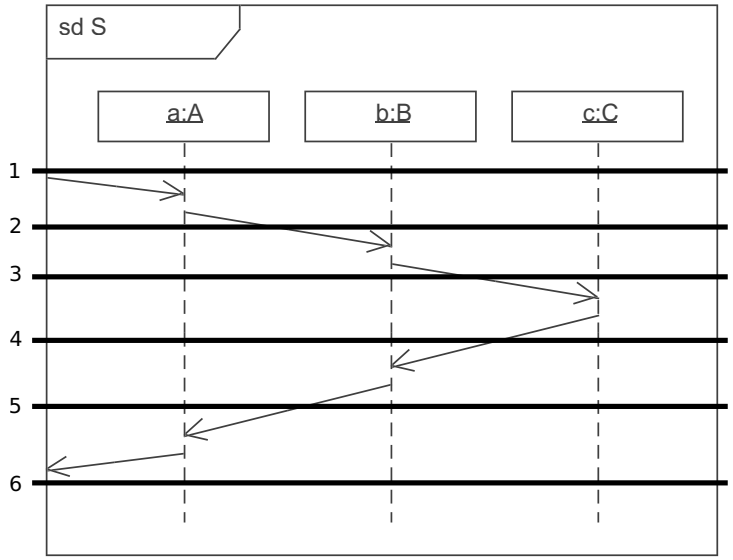


Figure 3.1: Sequence diagram with checkpoints marked as horizontal lines

state machine system outside the state machine the current transition is for. These are the events we must be able to discover, and log (*Event Effects* under section 3.2.1).

3.1 Framework Requirements

For our approach to back-in-time debugging, we require state machine systems that are making use of some state machine framework. We add the debugging functionality by making additions to the framework (and by modifying parts of the framework if necessary). For JFDebug, our state machine framework is JavaFrame. In addition to being capable of running state machine systems, as defined in section 2.4, our state machine framework must have scheduler and state machine concepts meeting the following criteria:

- A scheduler has a set of state machines that belong to it. It decides when each of these state machines get to execute its next transition.
- Every state machine in existence belongs to one scheduler; if a state machine does not belong to a scheduler, then it does, as far as the state machine system is concerned, not exist. And if it belongs to one scheduler it cannot belong to another one (at the same time).
- State machines can be moved between different schedulers during runtime.
- One transition can be executed at a time, per scheduler.

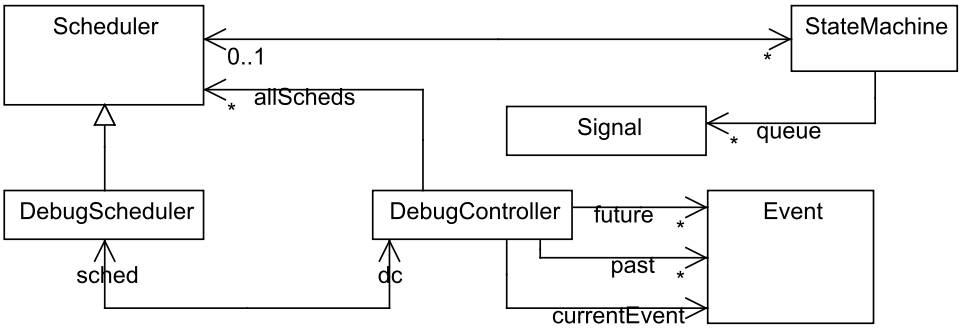


Figure 3.2: Debugger concepts

- For our debugger, we must be able to make our own scheduler implementation.
- Each state machine has a signal queue for incoming signals. An incoming signal remains in the queue until it triggers a transition.

While these are the main architectural requirements, there are more requirements. As we mentioned in the beginning of this chapter, if an event has an effect on the state machine system outside the state machine executing a transition, we must be able to discover it. And of course, the requirements depend on how willing we are to make changes to the state machine framework (or model transformation).

The scheduler. One of our requirements is that the framework has some kind of scheduler, that schedules state machine transitions, and one is that state machines can be moved between different schedulers during runtime. It should be noted that even without a scheduler like this most of our approach is still viable. What is *most* important is that we have clearly defined transitions, as we mentioned in the beginning of this chapter, and that we somehow are able to intercept the running system between these transitions.

The scheduler is part of the requirements because it lets us have one regular scheduler and one "debug" scheduler running at the same time. This makes it possible to put individual state machines into and out of debug mode during runtime by moving them between schedulers. This way, any performance overhead for the state machines that are not in debug mode should be minimal, because no debug functionality is added to the regular scheduler.

3.2 Overall Architecture

Figure 3.2 shows a conceptual model of our debugger. The *DebugController* keeps track of the schedulers in the existence and can move state machines

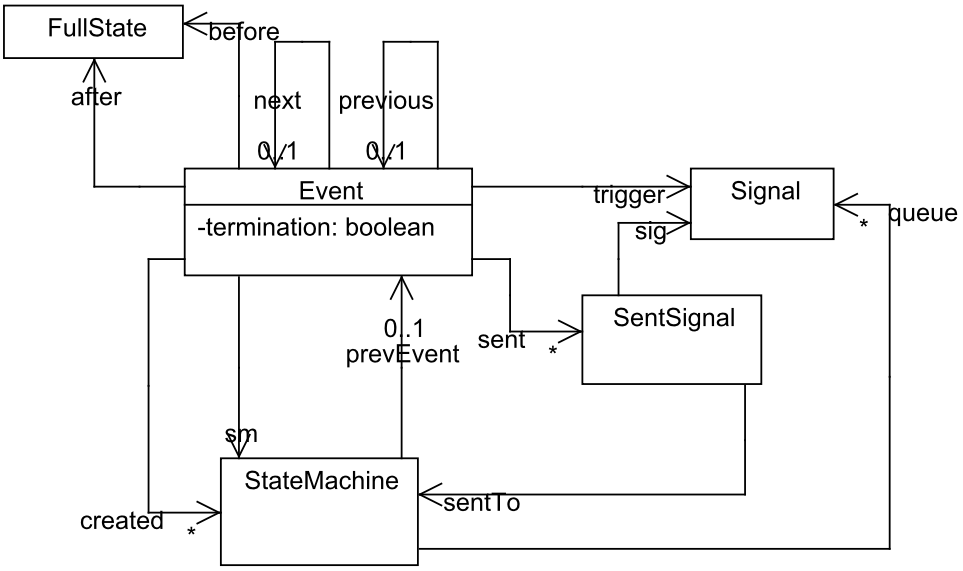


Figure 3.3: Our event model

to and from the *DebugScheduler*. The debug controller keeps track of the occurred events and the state machines in the debugged system, and it tells the debug scheduler when to pause and resume execution. *Scheduler*, *StateMachine*, and *Signal* are concepts we require from the state machine framework.

We define the set of state machines that belong to the debug scheduler as the **debugged system**. Anything outside of that is not part of the debugged system, and stepping in time will not effect it. And the user will only get detailed information about state machines in the debugged system. The only thing the debugger must be capable of doing outside the debugged system is to move state machines from other schedulers to the debug scheduler (and from the debug scheduler to other schedulers).

3.2.1 Event Model

Figure 3.3 shows an overview of our event model. One event is created for every occurred transition in the debugged system. Given a state machine, we find the most recently occurred event by following *prevEvent*. A *FullState* instance is a copy of the full state of a state machine. Copies are made before and after each event. *StateMachine* and *Signal* are concepts we require from the state machine framework.

Figure 3.4 shows an event sequence for 4 occurred transitions. The signals are not shown in the figure, and we will ignore those for now. The events have been created in the order *e1, e2, e3, e4*, which means that the transitions the events are for have occurred in that order. Similarly, the *FullState* instances have been created in the order *fs1, fs2, . . . , fs6*. Full states are copied

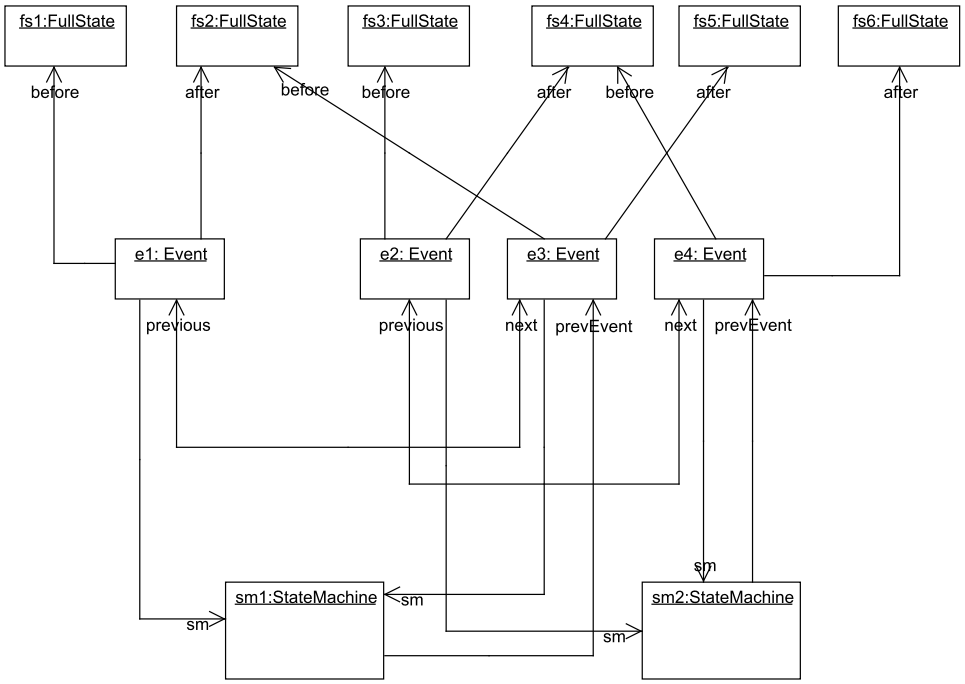


Figure 3.4: An event sequence

between events, so the event before and after one copy can use the same *FullState* instance. This means that the number of full state copies stored for one state machine is equal to the number of events occurred for that state machine plus one (or zero if no events have occurred for the state machine). And the number of full state copies stored for the state machine system is equal to the total number of occurred events, plus one for every state machine with any occurred events (in the figure: 4 occurred events plus 2 state machines with occurred events equals 6 *FullState* instances).

The event's *next* and *previous* make for a linked list of events for each state machine. This is not enough to determine the order of all the events. E.g. it is not enough to tell if *e1* or *e2* is the first event in the figure; we only know that *e1* is *sm1*'s first event and that *e2* is *sm2*'s first. For this reason, we also keep a list of all occurred events (the debug controller's *past*). If we want to discard events (for example old events to free up space, or new events because we are reverting the state machine system back to an earlier state), we must remove the event from this list as well as from the state machine-local linked list.

Event Effects

In addition to full state copies from before and after the transition, for each event we must store any occurred **event effect**. Any effect on the state

machine system that is not just modifying the state machine's full state is an event effect, and we must record it in order to be able to "undo" it when moving back in time. Event effects include sent signals and creation and termination of state machines:

- Signals sent during the transition are found in the event's *sent*. *SentSignal* is a *(Signal, StateMachine)* tuple with the signal sent and the state machine it was sent to: we need to remove the signal from the state machine's queue when undoing the transition.

- State machines created during the transition are found in the event's *created*. We must remove these state machines from the system when undoing the transition.

- We use the event's *termination* to keep track of state machines terminating. If it is set, we must put the state machine back into the state machine system when undoing the transition (possibly after recreating the state machine).

3.3 Recording Events

The debug scheduler will report to the debug controller before and after each transition. The debug controller will make a new event, *currentEvent*, before each transition. It will find the previous event for that state machine, by following the current state machine's *prevEvent*, and use that event's *after* full state copy for the new event's *before*. If no *prevEvent* exists for that state machine, a new full state copy is made for the new event's *before*.

During the transition event effects can be reported to the debug controller. If a signal is sent from the state machine executing the transition, to another state machine belonging to the debug scheduler, we create a new *SentSignal* instance and add it to the current event's *sent*. State machines created during the transitions are added to the event's *created*, and if the state machine terminates we set its *terminated*. After the transition, a new full state copy is made for the event's *after*. The event is then moved from the debug controller's *currentEvent* to its *past*, and the *prevEvent* for that event's state machine is updated so that it points to this new event. The event is now **finalized**. Once an event is finalized we will never add more event effects to it, or make changes to its full state copies. The only change we will make to a finalized event is to point its *next* to a new event the next time we create an event for the same state machine (or remove the event from *next* if the state of the debugged system is being reverted to a point in time between this event and its *next*).

3.4 Moving in Time

The debug scheduler must be capable of pausing and resuming execution of the state machine system. Pausing can only happen between transitions: if we're attempting to pause during a transition, the debug scheduler must wait until the transition has finished and the event for that that transition has been finalized and added to the controller's *past*, before pausing. We can move back and forward in time once the debugged system is paused.

In order to step one event back in time, we do the following:

1. Remove the most recent event from debug controller's *past*.
2. Revert the full state of that event's *sm* state machine to that of the event's *before* full state copy.
3. Undo any event effects.
   (a) For every *SentSignal* in the event's *sent*, we remove the signal *sig* from the signal queue *sentTo.queue*.
   (b) For every state machine in the event's *created*, we remove the state machine from the state machine system.
   (c) If the event's *termination* is set, we add the event's *sm* state machine to the state machine system.
4. Update the event's *sm* state machine's *prevEvent* so that it points to that event's *previous* event.
5. Put the event in the debug controller's *future*.

And we do the opposite for moving forward in time. We then move events from the debug controller's *future*, to its *past*, and we set the state machine's state to the event's *after* full state. We redo event effects, and we update the state machine's *prevEvent* so that it points to the event we are moving.

We define the **debugger's point in time** as the point in time just after the most recent event in the debug controller's *past*.

3.4.1 Internal and External Signals

Usually, if no signals are sent to the state machine system from outside of it, all state machines in the system will be inactive (no transitions will be executed). If we trace our way back in time from a transition, always finding the transition that sent the signal that triggered our current transition, we will end up with either a signal received from outside of the system, or a state machine's transition from its initial state (an initial transition). Initial transitions are caused by state machines being created, and so we can keep tracing our way back by finding the transition that created our current state

machine. In the end we will end up with a signal received from outside of the system, or the initial transition of one of the state machines that were created as the state machine system was started.

Inversely, we can say that a signal received from outside of the system causes a number of transitions to happen. One transitions must be triggered by the signal, and any number of transitions can be indirectly caused by the signal (i.e. all transitions that can be traced back, as above, to that signal).

We define **external signals** as signals sent from outside of the (debugged) system, to state machines in it. And we define **internal signals** as signals sent from state machines in the system to state machines in the system.

When stepping one event back in time, we "undo" the transition for that event. For a transition triggered by an internal signal, it is natural to put that signal back in the state machine's signal queue. There is a good chance that by reverting the state machine to the state it was in before the transition, we put it in a state where it is expecting that signal. If we do not put that signal back in its queue, we might be putting the state machine system in some state that could not possibly have been reached during a normal run. We call such a state for **impossible state** (of the state machine system). For example, a JavaFrame state machine's initial transition is triggered by a *StartMessage*, which is put in its queue during its creation. If we did not put the *StartMessage* back in its queue we would put the state machine in its initial state, expecting a *StartMessage* that it would never get.

Putting the signals back in the queues means that undoing a transition triggered by an internal signal does not achieve a whole lot. The signal will be put back in the queue, and when we resume execution of the state machine system the signal is likely to trigger the exact same transition again. Moving further back in time, until before the transition that sent that signal, we will remove the signal from the queue again (undoing the signal being sent). But if we then put the signal that triggered *that* transition back in that state machine's queue, resuming execution of the system will cause that transition to happen again, which will send a new signal triggering that transition we undid first.

However, we do not put external signals back in the queues when undoing transitions triggered by those. So by moving back in time to before an external signal was sent, we must have undone all the transitions that were indirectly caused by that external signal, and they will not happen again if we resume execution of the system. For example, by moving back in time to before a transition that as triggered by an incoming SMS and then resuming execution of the system, it will then be as if that SMS was never sent.

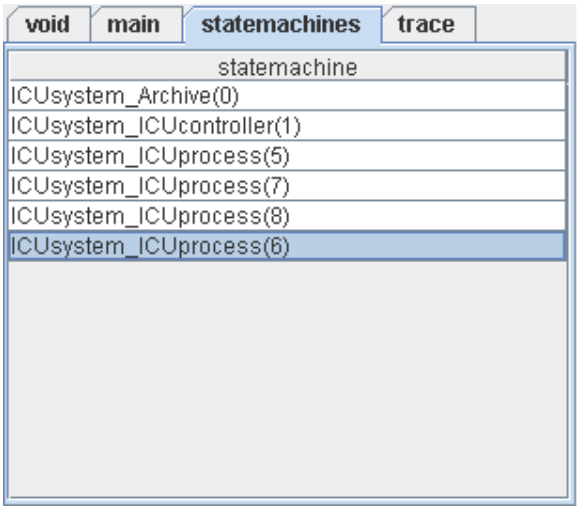


Figure 3.5: JFDebug's state machines tab

3.5 Presentation

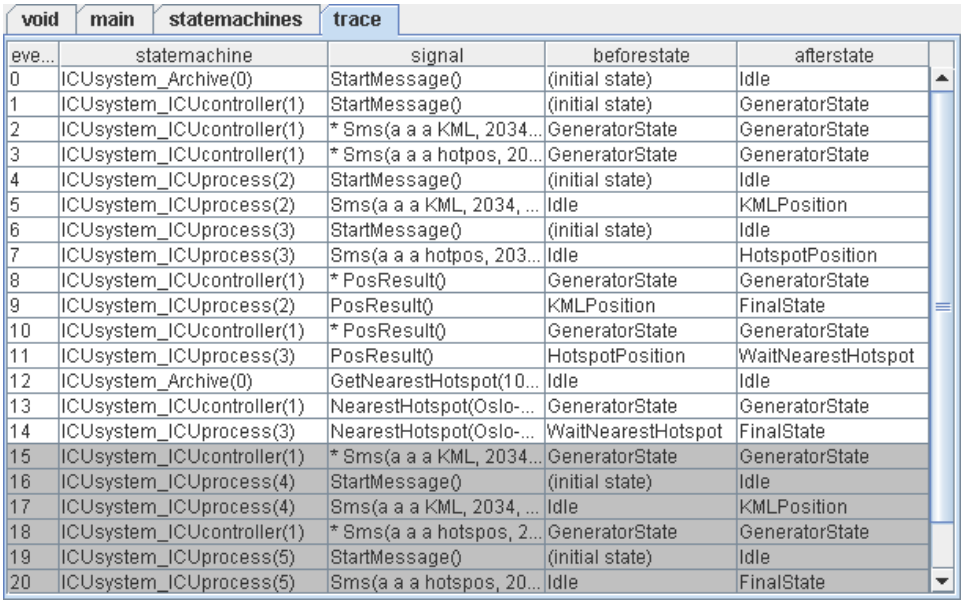
In this section we describe how information about the state machine system is presented to user, and how the user can navigate in time. We use screenshots of JFDebug as illustrations.

3.5.1 Inspecting the system

We have a few different ways of viewing information about the debugged system:

- A list of state machines in the debugged system.
- A trace of occurred events.
- A view for showing detailed information about one event.
- A trace of events visualized as a sequence diagram.
- Visualization of one event (using the modelling tool).

The list of state machines is updated as state machines are created and terminated and show the state machines currently in the system. It is updated as the system is executing, and as the debugger is used to move in time (it shows the state machines in existence at the debugger's point in time). We must be able to uniquely identify a state machines given its name. Figure 3.5 shows JFDebug's state machines tab. State machines are named in the format *<state machine type>(<unique id>)*.



eve...	statemachine	signal	beforestate	afterstate
0	ICUsystem_Archive(0)	StartMessage()	(initial state)	Idle
1	ICUsystem_ICUcontroller(1)	StartMessage()	(initial state)	GeneratorState
2	ICUsystem_ICUcontroller(1)	* Sms(a a a KML, 2034...	GeneratorState	GeneratorState
3	ICUsystem_ICUcontroller(1)	* Sms(a a a hotpos, 20...	GeneratorState	GeneratorState
4	ICUsystem_ICUprocess(2)	StartMessage()	(initial state)	Idle
5	ICUsystem_ICUprocess(2)	Sms(a a a KML, 2034, ...	Idle	KMLPosition
6	ICUsystem_ICUprocess(3)	StartMessage()	(initial state)	Idle
7	ICUsystem_ICUprocess(3)	Sms(a a a hotpos, 203...	Idle	HotspotPosition
8	ICUsystem_ICUcontroller(1)	* PosResult()	GeneratorState	GeneratorState
9	ICUsystem_ICUprocess(2)	PosResult()	KMLPosition	FinalState
10	ICUsystem_ICUcontroller(1)	* PosResult()	GeneratorState	GeneratorState
11	ICUsystem_ICUprocess(3)	PosResult()	HotspotPosition	WaitNearestHotspot
12	ICUsystem_Archive(0)	GetNearestHotspot(10...	Idle	Idle
13	ICUsystem_ICUcontroller(1)	NearestHotspot(Oslo-...	GeneratorState	GeneratorState
14	ICUsystem_ICUprocess(3)	NearestHotspot(Oslo-...	WaitNearestHotspot	FinalState
15	ICUsystem_ICUcontroller(1)	* Sms(a a a KML, 2034...	GeneratorState	GeneratorState
16	ICUsystem_ICUprocess(4)	StartMessage()	(initial state)	Idle
17	ICUsystem_ICUprocess(4)	Sms(a a a KML, 2034, ...	Idle	KMLPosition
18	ICUsystem_ICUcontroller(1)	* Sms(a a a hotspos, 2...	GeneratorState	GeneratorState
19	ICUsystem_ICUprocess(5)	StartMessage()	(initial state)	Idle
20	ICUsystem_ICUprocess(5)	Sms(a a a hotspos, 20...	Idle	FinalState

Figure 3.6: JFDebug's trace tab

Each entry in the event trace shows the name of the state machine that event is for, the signal that triggered the transition the event is for, and the state before and after the transition. The trace should shows events both from the *future* and the *past* when we are moving in time, but it must be easy to see where in the trace yhe debugger's point in time is. It must also be possible to tell external signals from internal signals. In figure 3.6 we see the JFDebug's trace tab. Each row represents one event. The events for rows with grey background are in the *future*, while the ones with white are in the *past*. External signals are marked with asterisks before the signal names.

An "event view" can be shown for any events. The event view shows the name of the state machine the event is for, the state and property values from before and after the transition the event is for, and the signal that triggered that transition. We use this to show information about state machines as well as individual events. A window with an event view can be opened for an event selected in the trace, or for a state machine selected in the state machines list. The event view for a state machine shows the most recent event for that state machine, given the debugger's point in time (so state and property values for that point in time are shown). Figure 3.7 shows JFDebug's event view. The name of the state machine it is for is shown at the top. Below it is the signal that triggered the transition the event is for, and below that is a table showing state and property values from before and after the event.

We record sent and received signals, and created and terminated state machines, and the events we have stored are ordered in time. This infor-

ICUsystem_ICUprocess(5)
Sms(a a a hotpos, 2034, aa-aaaaa)

	before	after
state	Idle	HotspotPosition
staticId	aa-aaaaa	aa-aaaaa
command		hotpos
fout		
lastpos_time		
parsedsms		[Ljava.lang.String;@2af0...
prstrm		
validpos	false	false
xcoord		
ycoord		
decxcoord	0.0	0.0
decycoord	0.0	0.0
filename		

Figure 3.7: JFDebug's event view

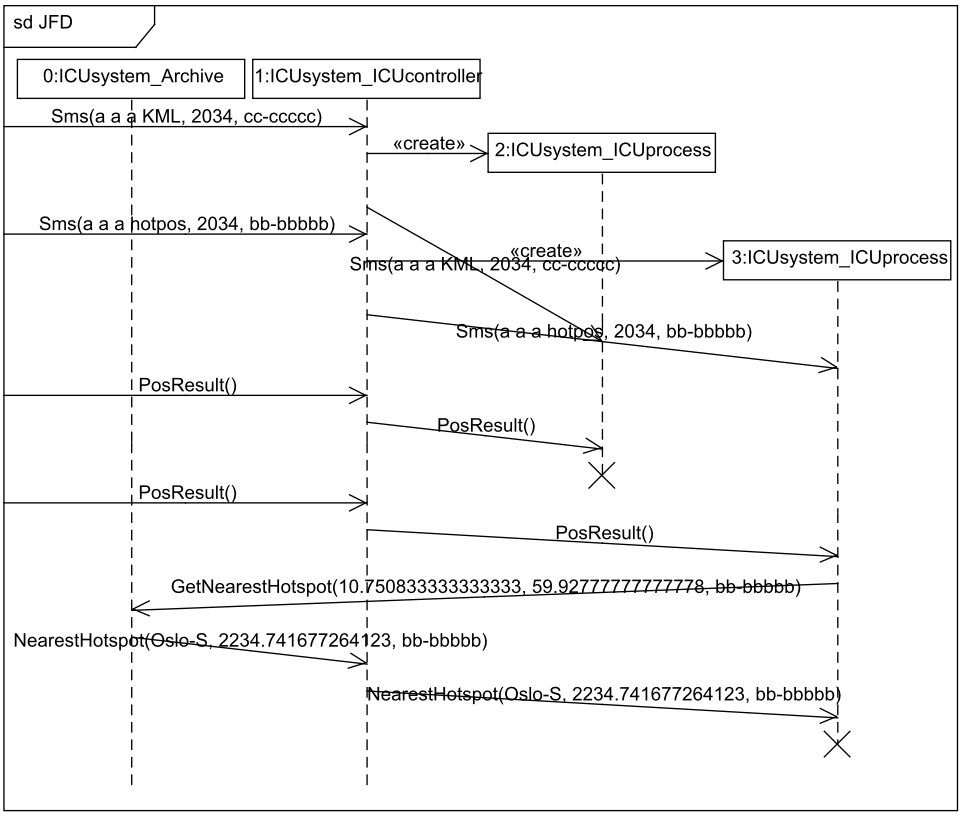


Figure 3.8: A trace of events visualized as a sequence diagram

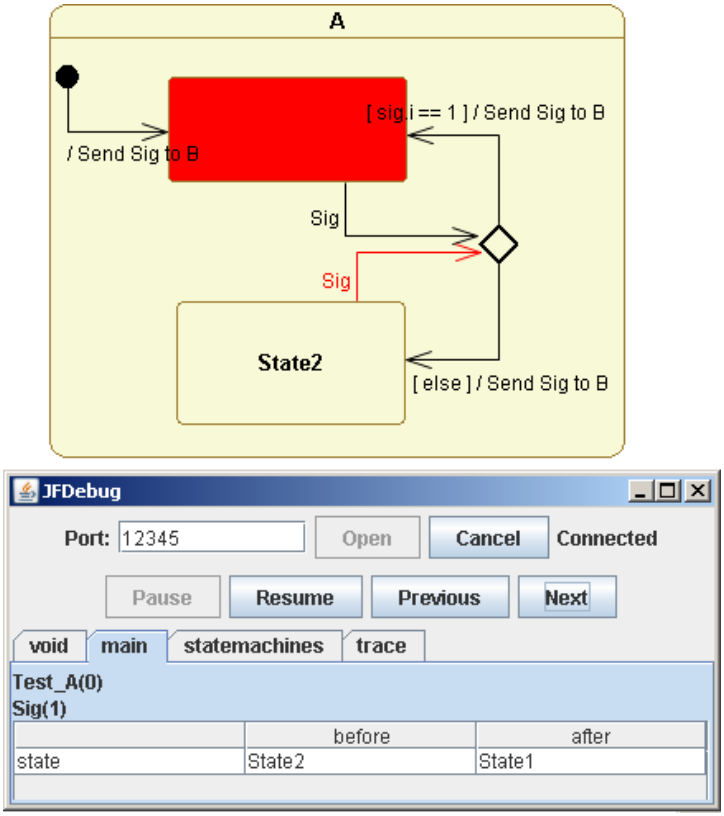


Figure 3.9: The event shown in the GUI is highlighted in the state machine diagram

mation can be used to visualize a trace of events as a sequence diagram. In JFDebug this is done by exporting UMLet diagrams. Figure 3.8 shows the sequence diagram exported from the trace of events shown in figure 3.6. The events from the *past* (white background in the trace tab) are shown in the sequence diagram. The signals sent from the edge of the interaction are external signals (the ones marked with asterisks in the trace tab).

Finally, we can use the modelling tool to visualize one event/transition. For JFDebug, we have made an Eclipse plug-in that highlights elements in Papyrus diagrams. Figure 3.9 shows an example (the main tab of JFDebug shows the most recently occurred event). In section 2.3.2 we discussed how JavaFrame transitions are related to UML transitions. With JFDebug we have taken a simple approach: we highlight the UML transition out of the "before" state that is triggered by the correct signal (the transition triggered by *Sig*, from *State2*), and we highlight the "after" state (*State1*; we cannot ses the "State1" text in the diagram because of the highlighting).

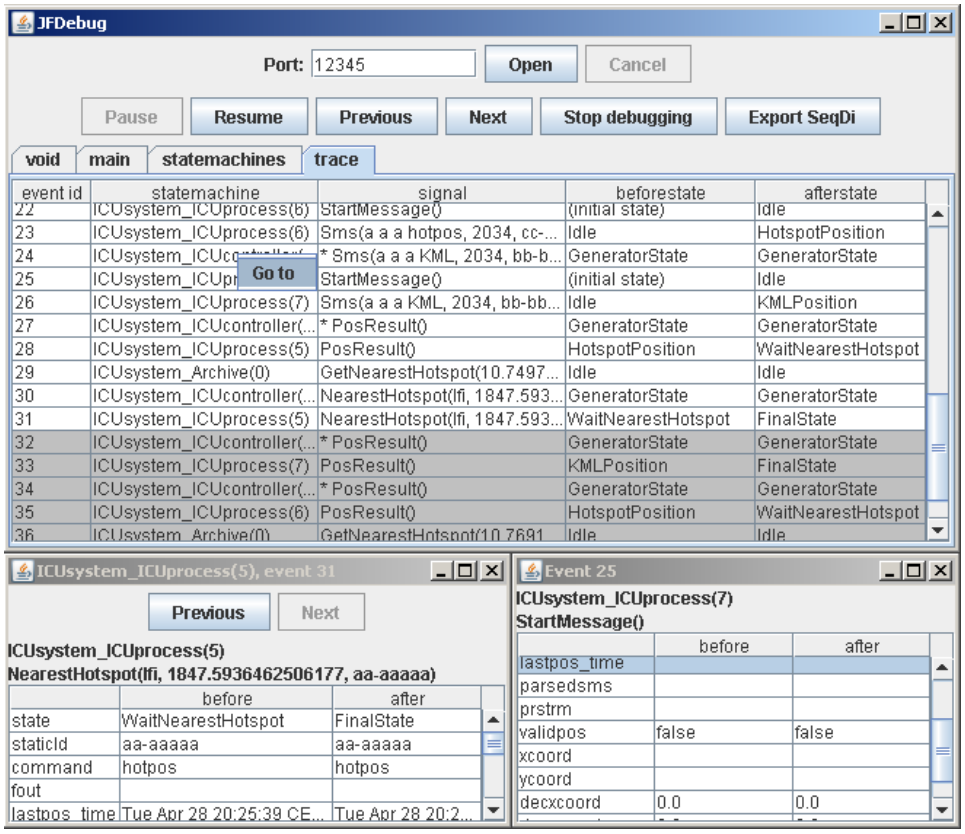


Figure 3.10: The JFDebug GUI in use, with one event window and one state machine window open

3.5.2 Moving in Time

We have three different ways of moving in time:

- We can step one event back or forward in time.
- We can move to the point in time where a selected event is the most recently occurred one. The event is selected from the trace.
- For a given state machine, we can move back or forward in time until that state machine's most recently occurred event changes.

Figure 3.10 shows the JFDebug GUI in use. The main window is on the top, showing the trace. Below it is a state machine window to the left (opened by double-clicking an entry in the state machines list), and an event window to the right (opened by double-clicking an entry in the trace). Both the state machine window and the event window show event views. In the main window of the GUI, there are "Previous" and "Next" buttons, for

stepping one event back or forward in time. An entry in the trace has been selected and then right-clicked; by clicking "Go to" the debugger will move to the point in time where that event is the most recently occurred event. Finally, the state machine window has "Previous" and "Next" buttons for moving back or forward in time until the most recently occurred event for that state machine (its *prevEvent*) has changed.

The user can open as many state machine windows and event windows as he would like, and all state machine windows are updated as the debugger moves in time. The user can make use of the three different ways of moving in time to get to a desired point in time, or to move back and forward in time. And while doing so, to keep an eye on the state of the debugged system, he can have state machine windows open for any state machines of interest.

Chapter 4

Creating the Tool

This chapter describes the design and implementation of the tool JFDebug and an Eclipse plug-in for Papyrus that can be used together with it. JFDebug is created for JavaFrame, and is intended to be used with the systems compiled with JavaFrame Transformation.

4.1 Design Goals and Priorities

In order of importance, these are the main features our debugger should have:

- The debugger should let the user pause the JavaFrame system under debugging, move back and forward in time once it is paused, and resume execution of the system from an earlier point in time. This functionality should be achieved by storing the full states of state before and after transitions and by recording important events during transitions, as described in chapter 3.
- The debugger should present information about the debugged system to the user.
  - The information presented should be relevant to the debugger's point in time. When we have paused the system and are moving in time, the information should reflect the state the system was in at the point in time we have moved to. And when the system is executing, the information presented should reflect the current state of the system.
  - The information presented should be model-level. It should deal with state machines, transitions, signals, etc., rather than lower level details (method calls, Java statements, ...). It should be understood by the modeller of the system, even if he does not

have a good understanding of the Java code that his model was transformed to or of the Java code that JavaFrame consists of.

- The debugger should be capable of making some connection to the model that the debugged system was transformed from and the modelling tool that was used.

The debugger is used by using our new JFDebug library instead of the regular JavaFrame library. When we're adding new functionality to our debugger, we try to follow a couple of guidelines:

First, we do not want to make changes to JFT. We want the debugger to work for systems made for, and transformed by, JFT, rather than require some special "debug" transformation. We want the same Java code to run and behave the same way whether our JFDebug library or the regular JavaFrame library is used.

Second, we do not want to make changes to the classes that already exist in regular JavaFrame. We want to be able to switch between debug mode and production mode during runtime, and we want to cause minimal overhead when in production mode.

New functionality, then, should be put in new classes added in our JFDebug library, and this new functionality should be optional. It should be possible to use the old classes instead, for all of, or parts of, the JavaFrame system. For example, we add a `DebugMailBox` class that reports received signals instead of adding this to the `MailBox` class that already exists in regular JavaFrame. State machines can be part of the debugged system and use the debug mailbox, or run in production mode and use the regular mailbox.

We break these guidelines a few times. For example, we do make a few changes to the regular mailbox in order to be able to override some methods in our debug mailbox subclass. This does mean that using our JFDebug library will introduce *some* overhead over using the regular JavaFrame library, even if we're only running in production mode. In the end, our goal is to keep that overhead as low as possible, but still choose good solutions for the new functionality we are adding.

Finally, the tool is made to be used in the *INF5150 – Unassailable IT-systems* course. The JavaFrame systems made in the course are fairly small and do not usually run for long enough to reach large numbers of transitions. While the core of the debugger should allow us to have only parts of the JavaFrame system running in debug mode, this functionality is not important for the students of the course: the entire system can just run in debug mode all the time, without ever encountering any scalability problems. Also, the code generated by JFT does not make use of more than one scheduler, so in JFDebug we do not take the possibility of more than one regular JavaFrame scheduler being used into account.

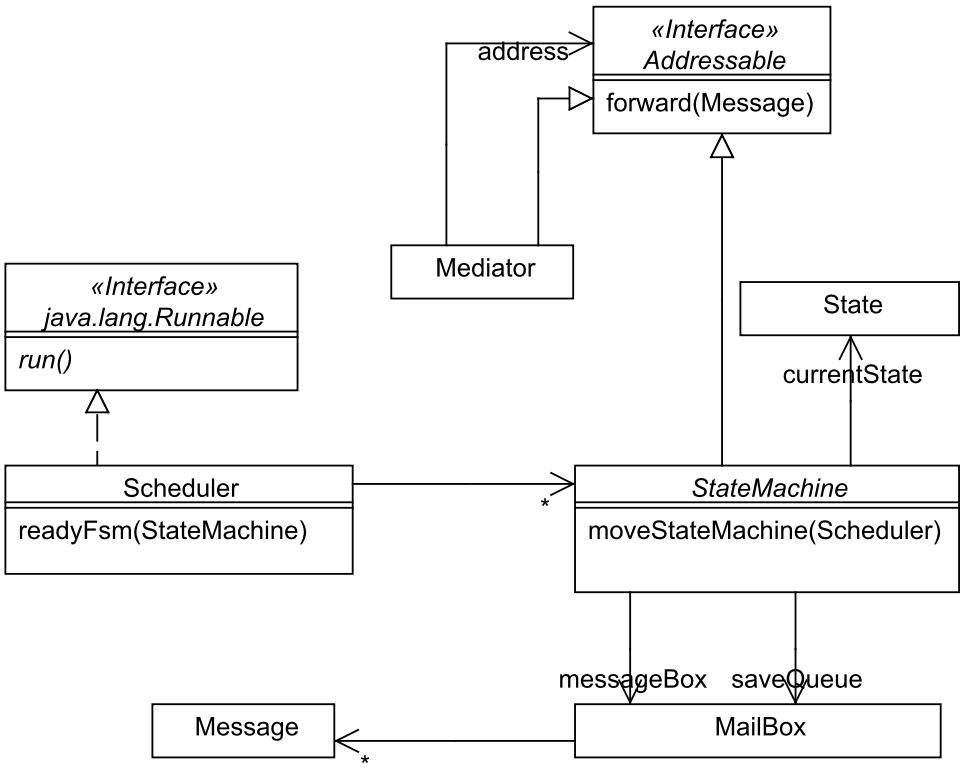


Figure 4.1: JavaFrame overview

4.2 JavaFrame

JavaFrame was introduced in section 2.3.1. What it is and how it is used is explained there. This section is about how JavaFrame, works (rather than how systems using JavaFrame works), and describes technicalities about its implementation that are relevant to design and implementation decisions made when creating JFDebug.

Figure 4.1 shows an overview of JavaFrame. When a JavaFrame system is running, one or more `Scheduler` objects are handling state machine transitions. Each scheduler is running in its own thread, and each scheduler has a list of state machines that are waiting to execute transitions.

The state machines are objects of subclasses of the abstract class `StateMachine`. Every state machine in the system belongs to one scheduler, and can be moved between schedulers with the `moveStateMachine` method. Each state machine has two `MailBox` objects. The `messageBox` contains received signals (i.e. `Message` instances) that are not yet processed. This is the state machine's *signal queue*. When a signal is received, the state machine will put it in its `messageBox` and then call its scheduler's `readyFSM` method. `readyFSM` will put the state machine at the end of the scheduler's list of state

machines, or do nothing if the state machine is already in the list.

The `saveQueue` is used for *deferred signals*. Each state can defer a number of signal types. If the state machine is in a state that defers any signal types, signals of those types will be put in the `saveQueue` when they are triggering transitions. Whenever a state machine enters a new state, all signals its `saveQueue` are put in its `messageBox` again.

When the system is running, each scheduler runs in a loop where it handles state machine transitions. In each iteration of the loop, the scheduler will pick the first entry in its list of state machines and remove it from the list. It will then remove signals from the state machine's mailbox, and call the state machine's `exec` method for each one.

Moving on to the next state machine in the list takes some time (one iteration of the loop), so the more signals are processed for each state machine before moving on to the next one, the faster the system runs (less overhead per signal processed). The scheduler will process several signals for each state machine, before moving on. To ensure that the scheduler is fair, that all state machines eventually get their signals processed, there is a maximum number of signals that will get processed this way in each iteration of the loop (30 in the current version of JavaFrame). If there are still signals left in the mailbox after this, the state machine is put at the end of the scheduler's list. The state machine makes sure it is added to the list again if it receives a new signal, so if there are no signals left in the `messageBox`, the scheduler does not need to add it to its list of state machines.

4.3 JFDebug Architecture

Figure 4.2 shows an overview of the JFDebug. As JFDebug is used as a replacement for JavaFrame, it includes its own versions of the regular JavaFrame classes (`Scheduler`, `StateMachine` and `MailBox` in the figure). Because the debugger needs to subclass the `Scheduler` and `MailBox` classes, some modifications are made to these. In these classes some fields that are `private` in regular JavaFrame are `protected` in JFDebug, so that they can be accessed in subclasses. And some methods that are `final` in regular JavaFrame are not in JFDebug, so that they can be overridden. In addition, the `StateMachine` class has a new `StateMachineMetaData` field. No changes are made so that the JFDebug versions of JavaFrame classes no longer work as described in the previous section.

The `DebugScheduler` handles execution of a JavaFrame system in the same way as a regular JavaFrame shceduler does, and is also capable of pausing execution of the system. Both the `DebugScheduler` and `DebugMailBox` reports information about events to the `DebugController`. The controller creates `Event` objects using the information that is reported to it from the debug scheduler and mailboxes, and keeps track of all the events. It also offers

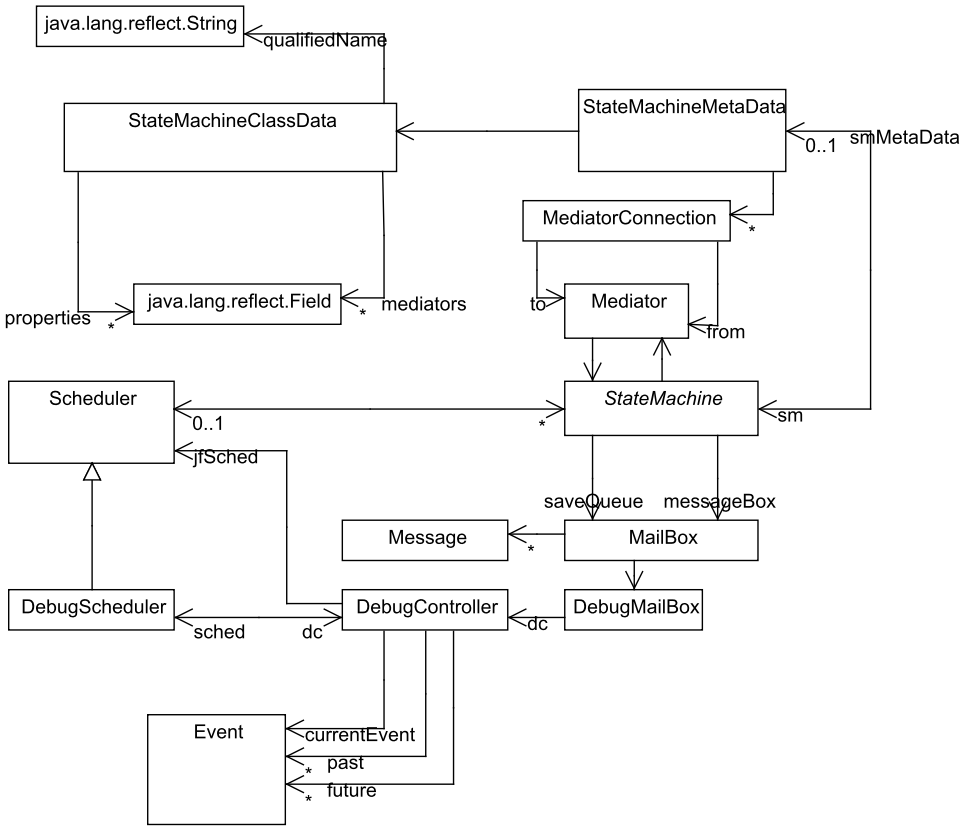


Figure 4.2: JFDebug concepts

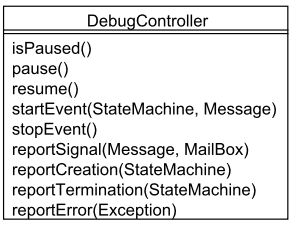


Figure 4.3: Important *DebugController* methods

functionality for pausing, resuming, inspecting state machines and events, and starting and stopping debugging. The GUI lets the user of the debugger use the functionality offered by the controller.

Scheduler. JFDebug has its own scheduler, `DebugScheduler`, in addition to the (slightly modified) `Scheduler` from regular JavaFrame, which the debug scheduler is a subclass of. With the `moveStateMachine` method, state machines can be moved between the regular scheduler and the debug scheduler. As we said in section 3.2, we define the state machines belonging to the debug scheduler as the debugged system. Anything outside of that is not, whether it is a state machine belonging to another scheduler or another thread (such as a thread that handles incoming SMSes).

The debug scheduler handles state machine transitions the same way as the regular scheduler. In addition it will ensure that information about state machines is recorded between each transition, and have functionality for pausing and resuming execution of the debugged system.

Mailboxes. JFDebug also has its own mailbox, `DebugMailBox`, which is a subclass of the `MailBox` class from JavaFrame. All state machines in the debugged system will use debug mailboxes instead of regular mailboxes. In addition to offering the functionality of a regular mailbox, a debug mailbox will ensure that information about received signals is recorded.

`DebugScheduler` and `DebugMailBox` are the only classes from JavaFrame that we have "debug" subclasses of. For all others, we just use the regular JavaFrame classes, no matter if we are in debug mode or not.

Controller. The `DebugController` class handles recording of events and stepping in time. Using information reported from the scheduler and mailboxes, the controller makes `Event` objects. By pausing and resuming the scheduler, and by making use of the information recorded in the `Event` objects, the controller takes care of moving in time. Some important methods of the debug controller, that will be referred to throughout this chapter, are shown in figure 4.3.

Metadata objects. Instead of adding several new fields to the `StateMachine` class (like in figure 3.2 on page 16), we use a `StateMachineData` for keeping all the additional information about state machines required by the debugger. As we said in section 4.1, we prefer to add new classes rather than modifying existing classes. By using these metadata objects we only need to add one new field (`smMetaData`) to JavaFrame's state machine class. There is one such metadata object for each state machine in the debugged system. Each of these objects keeps a pointer to

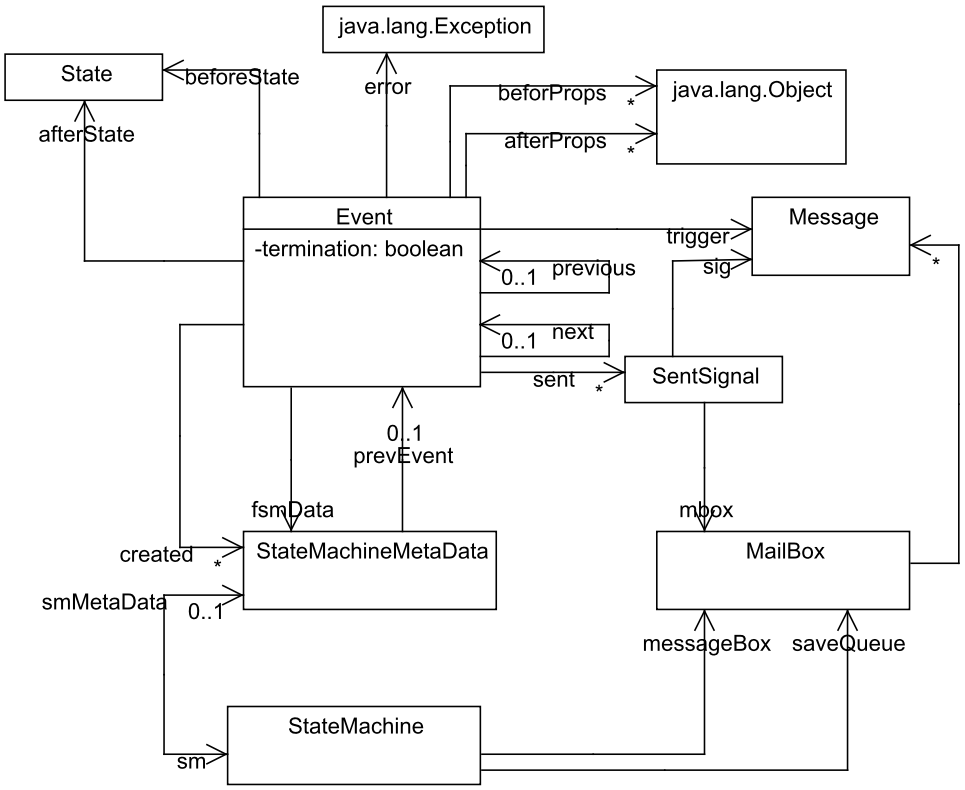


Figure 4.4: JFDebug event model

the most recent event for its state machine, and as a list of all mediator connections that are used to send signal to its state machine.

For every type of state machine there is in the system (for every subclass of `StateMachine`), there is a `StateMachineClassData` object. Each of these class data objects contains two arrays of Java `Field` objects: one for all the mediators and one for all the properties of its state machine type. The mediator `Field` objects are used by `StateMachineData` objects when searching for mediator connections, and the property fields are used when copying its state machine's full state.

4.4 Events

Execution of the debugged system is recorded as a series of events. There is one event for every JavaFrame transition (every signal processed by a state machine). The event model, as implemented in JFDebug, is shown in figure 4.4. Moving in time is done by stepping through the recorded events, sor each event we're interested in storing any information required to undo the JavaFrame transition that event is for. In addition, if any error occurs

during the transition, we want to store information about it. For each event, we store the following:

- The full state of the state machine, before and after the event. This includes:
  - The state the state machine was in. This is kept in its `state` variable. We store it in the event's `beforeState}`/`afterState`.
  - The values of the state machine's properties. While both UML properties and ports are transformed to fields in the state machine's class, all ports are transformed into mediator type variables. So the properties are all the fields declared in the state machine's class that are not mediator types. We store them in the event's `beforeProps`/`afterProps`.
- Event effects:
  - Signals sent during the event (the event's `sent`).
  - State machines created during the event (the event's `created`).
  - Whether or not the state machine terminated during the event (the event's `termination`).
- Errors. Exceptions can be thrown during a transition. These are handled by the regular JavaFrame scheduler, and should be handled the same way in the debug scheduler. But they should also be stored so that it can be viewed when inspecting that transition's event. We store them in the event's `error`.

Events are created as the debug scheduler and debug mailboxes report information about events to the debug controller. The controller takes care of creating and keeping track of the event objects.

Section 4.4.1 is about we are create and finalize events. In addition to creating new events, this takes care of recording the full state of the current state machine before and after each of event/transition. How we report event effects and errors is described in section 4.4.2.

4.4.1 Creating and Finalizing Events

The debug scheduler calls the debug controller's `startEvent` method before each signal is processed, and `stopEvent` after. `startEvent` makes the controller create an `Event` object and make sure that the full state of the state machine before this event is stored. The controller will find the most recent event recorded for the same state machine, and use that event's `afterState` and `afterProps` for the new event's `beforeState` and `beforeProps`. If no

previously recorded events exist for the same state machine, new full state data is copied from the state machine instead.

Between a `startEvent` and `stopEvent` call, event effects (signals sent, state machines created and state machine termination) can be reported to the debug controller to be recorded as part of the current event.

`stopEvent` finalizes the current event. State data is copied from the state machine and stored in the event's `afterState` and `afterProps`. The event is added to the debug controller's `past`, and event effects can no longer be added to it. Event effects reported are ignored until `startEvent` is called and a new event is created again.

Copying State

The full state, that is the values of the runtime equivalents of the UML state and properties of the state machine, are copied between every JavaFrame transition for every state machine in the debugged system. When we copy the full state, it is important to make sure that our copies do not point to data that is going to be modified as the system is running; we want *deep* copies of all mutable data. At the same time, we always want the debugger to let the debugged system keep running as it normally would. The debugger should try to get a deep copy, but if it cannot it should make do with what it can do, and let the debugged system keep executing. It should, for example, never throw an exception.

Copying the UML state is simple. The `State` objects are not mutated during execution of the system, and we can just copy the current (pointer) value from the state machine's `state` variable. To copy the UML properties of a state machine, we use the `properties` of the `StateMachineClassData` for the current state machine's type. For each field in the `properties` array, we retrieve its value from the current state machine. For each field/value we attempt the following:

1. If the value is `null`, we store `null`.

2. If the field type is a Java primitive, or if the value is an instance of `String` or of one of the primitive wrapper classes (`Integer`, `Double`, etc.), we copy the value/pointer. The `String` class and the wrapper classes are immutable.

3. Instances of JavaFrame's `Message` class can be copied by calling `Message`'s `duplicate` method: if the value is an instance of `Message`, we call `duplicate`.

4. If the value is an instance of `Serializable` (a class implementing the `Serializable` interface), we write the object to a byte stream, then read from that byte stream to get a copy.

5. If the object is an instance of a `Cloneable` class, we try calling its `clone` method.

6. We store a pointer to the object.

The different approaches to copying a value are tried in the order given above. For every value we're copying, we go through the list until we succeed. E.g. we will never try to write an `Integer` object to a byte stream, but if writing a `Serializable` object to a byte stream throws an exception, we will see if we can clone it instead, and so on.

Step 5 and 6 are the least preferred ones. Serializing an object is preferred over calling `clone`, as we do not know if a `clone` method returns a deep or shallow copy. Step 6 just ensures that we always copy something; if we copy a pointer it is quite likely that the object it is pointing to gets mutated as the debugged system executes.

4.4.2 Event Effects and Errors

Between calls to `startEvent` and `stopEvent` event effects and errors can be reported. The debug scheduler can report to the debug controller if state machines are created, if the current state machine is terminated, or if an exception is thrown from the transition. And debug mailboxes can report to the controller when they receive signals.

The debug scheduler can discover state machines being created and terminated in its `readyFSM` method. And by putting the execution of a transition in a try/catch block, exceptions can be caught and reported. Messages sent are reported from the `addMessage` method of the debug mailbox.

State machine creation. After the creation of a state machine, its scheduler's `readyFSM` is called with the new state machine as argument. `readyFSM` is also called when a state machine is moved to a scheduler and when a state machine has received a signal. It is important to not mistake other calls to `readyFSM` for state machines being created.

A newly created state machine has a signal of type `StartMessage` in its mailbox. As it can receive new signals before its start signal is processed, `readyFSM` may be called more than once while the start signal is in its mailbox. The first time a state machine calls `readyFSM`, its mailbox will contain only one signal, while for subsequent `readyFSM` calls caused by received signals it will contain more than one signal. The debug scheduler checks that the first signal in the mailbox is a start signal and that the first and last signal in the mailbox is the same one (i.e. there is only that one signal in the mailbox) before reporting it by calling the debug controller's `reportCreation`. `reportCreation` adds the `StateMachineMetaData` object for the new state machine to `currentEvent.created`.

As we said, `readyFSM` calls can also be caused by the state machine being moved to the debug scheduler. We can use the debugger to move state machines to and from the debug scheduler, so it is possible for a state machine to be moved to the debug scheduler while it happens to have just a `StartMessage` in its mailbox. If that happens, `reportCreation` gets called at the wrong time. We also call `readyFSM` when we are moving back in time (as we are putting signals back in their queues, the state machines must be put in the scheduler's list of state machines waiting to execute transitions). Navigating back to when the state machine had only the start signal in its mailbox will cause a `reportCreation` call. However both moving state machines to the debug scheduler and navigating in time can happen only when the debugged system is paused. When the system is paused, no transition is executing, and the debug controller has no `currentEvent`. And when there is no `currentEvent`, the debugger will ignore any reported event effects.

State machine termination. On reaching its final state and terminating, a state machine will call `moveStateMachine(null)`. `moveStateMachine` is used for moving a state machine to a scheduler. If the state machine already belongs to a scheduler, it will set its `wantedScheduler` before calling `readyFSM`. In `readyFSM` the debug scheduler checks if the state machine's `wantedScheduler` variable is `null` and calls the debug controller's `reportTermination` method if it is. `reportTermination` sets `currentEvent.termination` to `true`.

Signals. All state machines in the debugged system use debug mailboxes instead of regular mailboxes. Signals received by debug mailboxes are reported to the debug controller by calling the method `reportSignal`. Signals that are sent from a state machine in the debugged system but are not received by one in the debugged system will not be reported and recorded as part of the event. Moving back in time should not have effects outside of the debugged system, so we are not interested in "undoing" signals sent out of it, and we do not need to store them as parts of our events.

We need to tell internal signals from external ones (section 3.4.1). The debug scheduler has its own thread and all signals sent from state machines within the debugged system will be sent in this thread. The debug controller keeps a pointer to the Java `Thread` that the debug scheduler runs in, so when a signal is reported to the controller, it can compare this pointer to the current thread to check if the received signal is an internal or external signal.

Internal signals must have been sent as a part of the current JavaFrame transition and are added to the current event, while external signals are not. For internal signals we create a `SentSignal` instance and add it to the

event's `sent`. `SentSignal` has pointers to both the signal and the mailbox that received it, so that we can remove the signal from that mailbox when moving back in time.

Errors. If an exception is thrown from a JavaFrame transition, the debug scheduler must report it to the debug controller. The controller will then make sure that a pointer to the exception is added to the current event. If an exception is thrown, it has no effect outside the current state machine. So reverting the full state of the state machine to what it was before the erroneous transition does a "recovery" from the error. We do not need to know about the exception to be able to move back in time.

The reason why we do store exceptions is that for so that we can present it to the user of the debugger. Events that caused exceptions can easily be spotted in the trace tab of the GUI, and the stack trace for the exception will be shown when inspecting one of those events.

Catching and reporting events is straightforward. In the debug scheduler, the call that executes one JavaFrame transition is put in a `try` block, with a `catch` block catching any exceptions. Caught exceptions are reported by calling the controller's `reportError` method, adding the exception to the current event. `reportError` calls adds the exception to `currentEvent.error`. After reporting the error, the debug scheduler throw the exception so that it is handled the same way the regular JavaFrame scheduler handles exceptions.

4.5 Moving in Time

In order to move in time, the debugger must be capable of pausing the debugged system between transitions. And it must be capable of stepping back and forward through events when the debugged system is paused.

4.5.1 Pausing Execution

The debugged system should be paused as soon as possible after the user has pressed the pause button. Pressing this button results in a call to the debug controller's `pause` method. This method must make sure that the debug scheduler knows that it is supposed to pause, and then wait for the scheduler to pause before returning.

This will happen in a GUI thread, while the scheduler runs in its own thread. The controller will set a `paused` variable that can be read by the scheduler. The scheduler checks this variable between each transition, and stops executing transitions if it is set. The scheduler also has its own `paused` variable; the scheduler can check the controller's `paused` to see if the user wants execution to pause, while the controller can check the scheduler's `paused` to check if the execution is paused yet.

The Java `synchronized` keyword and the `wait` and `notify` methods[1] are used to make the controller wait for the scheduler to pause. The thread calling the controller's `pause` method acquires a lock with `synchronized`, then sets its `paused` variable to `true`, before calling `wait`. `wait` makes the thread release the lock wait until another thread calls `notify`, and then acquire the lock again before resuming. The debug scheduler acquires the same lock and checks the controller's `paused` between transitions. When the scheduler sees that the controller's `paused` is set, it sets its own `paused` to true before calling `notify` (waking up the controller) and then `wait` (releasing the lock so that the controller can resume). Now the controller will check the scheduler's `paused` variable, and return if it is set to `true`. If for some reason `notify` is called, or `wait` throws an exception, and the scheduler's `paused` is (still) `false`, the controller will just call `wait` again. It is guaranteed that execution of the debugged system really is paused by the time the `pause` method returns.

4.5.2 Stepping through Events

The debug controller keeps two lists with `Event` objects, `future` and `past`, as well as one pointer to the current event. As the debugged system is running, new event objects are created, finalized and added to the beginning of the past list. Calls to `startEvent` creates a new current event, while `stopEvent` moves the current event to the past list. The future list remains empty while the debugged system is running.

Once the debugged system is paused, we can step back and forward in time. When stepping back in time, event objects are moved from the beginning of the `past` to the beginning of the `future`, and when stepping forward in time they are moved from `future` to `past`. The first event in the `past` list is the most recent event for the debugger's point in time, and the first event in the future list is the first event that occurred after that point in time. The two lists extend in opposite directions from the debugger's point in time.

When stepping back and forward like this, we do not make changes to the actual state machine objects. Instead we only change the `StateMachineData` objects in order to be able to give a view of the system at the desired point in time. Whenever we make a step back or forward in time, we update the `prevEvent` variable of the `stateMachineData` object for the current state machine. When inspecting the state machine with the GUI, only information from this current event is used; we do not need to modify the actual state machine object until we resume execution of the debugged system.

[1] `http://java.sun.com/docs/books/tutorial/essential/concurrency/guardmeth.html`

4.5.3 Resuming Execution

`synchronized`, `wait` and `notify` are used by the scheduler to wait until the user wants the execution to resume. When the system is paused, the debug scheduler thread is waiting. The controller's `resume`, called by a GUI thread, acquires the lock and sets the controller's `paused`. It then calls `notify`, releases the lock, and returns. Once the scheduler is notified and wakes up, it acquires the lock and checks the controller's `paused`. If it is `true` it calls `wait` again, and if it is `false` it sets its own `paused` to `false` before releasing the lock and resuming execution.

When we do resume execution of the debugged system, the state machine objects in the debugged system must be reverted to the point in time we have moved to. We must undo the event effects and full state changes caused by the transitions for all the events in the `future`, starting with the event that occurred latest in time. We do this by removing the event at the end of the future list and calling its `undo` method, and we repeat this process until the future list is empty. `undo` does the following:

- Copy the the events `beforeState` and `beforeProps` to the state machine the event is for.
- Remove signals sent during the event's transition from the debug mailboxes that received them. For each `SentSignal` in the event's `sent`, remove `sig` from `mbox`.
- Remove state machines created during the transition (the event's `created`) from the JavaFrame system. Remove pointers to their mediators from all other mediators in the system and remove them from the debug scheduler.
- If the state machine terminated during the transition (if its `terminated` is `true`), add it to the JavaFrame system again. Recreate the pointers to its mediators that were removed when the state machine was removed from the JavaFrame system, and move the state machine back to the debug scheduler.
- If the transition was triggered by an internal signal, put the triggering signal back in the state machine's mailbox.

4.6 GUI

We have shown what JFDebug's GUI looks like, and explained what it is used for in section 3.5. In this section we will look at the interaction between the GUI and the debug controller; what is reported to the GUI and what functionality is exposed to the GUI.

The debug controller report events of interest to the GUI:

- While the debugged system is running, when a new event is finalized it is reported to the GUI.
- While the system is paused, when stepping one event back or forward in time, it is reported to the GUI along with the most recent event for our (new) chosen point in time.
- When a state machine is moved from the regular JavaFrame scheduler to the debug scheduler, it is reported to the GUI.
- When state machines are created and terminated, it is reported to the GUI. It is also reported when stepping through events where state machines were created or terminated.

When an event is reported, the main tab of the GUI and any open state machine windows for the current state machine are updated with information about the new event. And the trace tab is updated: new rows are added as new events are reported (i.e. during execution of the debugged system), and events are coloured grey/white as steps back and forward in time are reported. As creation and termination of state machines are reported, rows are added and removed in the state machines tab of the GUI.

The debug controller offers functionality that the GUI uses:

- Navigation:
  - `previous` and `next` methods for stepping one step back/forward in time.
  - `previous` and `next` methods taking `StateMachineMetaData` arguments, to step back/forward in time until the most recent event of the given state machine has changed.
  - A `gotoEvent` method, taking an event number argument.
- Inspecting the debugged system:
  - A `getFsm` method, taking the id of a state machine as argument, returning the `StateMachineMetaData` object for that state machine.
  - A `getEvent` method, taking an event number argument, returning the `Event` object for that event.

The arguments given to the `gotoEvent` and `getEvent` methods are numbers relative to the current position in time. 0 is the most recent event (making `gotoEvent(0)` a no-op), 1 is the first event from the `future`, -1 is the second event from the `past`, etc.

4.7 The Eclipse Plug-in

The debugger runs fine without being connected to the Eclipse plug-in we have created for it. The debugger will behave the same way whether it is connected to the plug-in or not; the only difference will be that once it is connected, elements in Papyrus diagrams are highlighted as the debugger is used.

The plug-in depends on the JFT plug-in for Papyrus, and uses the same method JFT uses to generate Java names from UML qualified names. Before connecting to the debugger, a UML package must be selected (the same that was used for transformation of the debugged system), and the plug-in will search through the UML model and gather qualified names and Java names for elements of interest. When connected to the debugger, the qualified names and the Java names are sent to the debugger, making it possible for the it to know the qualified name for the UML equivalent of anything that has been transformed from the UML model. Given a state machine, state, composite state/region, signal or regular Java class, the debugger can find the correct UML qualified name.

In the work with [18], Bjørn Brændshøi developed a routine for highlighting elements in Papyrus diagrams. Given a set of UML qualified names, diagram elements are highlighted. Our debugger uses that routine, with some minor modifications (removing dependency on Brændshøi's own plug-in), to highlight states and transitions in Papyrus diagrams. The qualified name for the most recent transition's triggering signal and its "before" and "after" states are sent to the plug-in as new events are finalized, and as we move back and forward in time. Given the signal and "before" state, the plug-in will find the UML transition from that state, triggered by that signal. The qualified names for this and the "after" state is then used for the highlighting.

As the connection to the modelling tool is not yet being used for very much, what is most interesting about it is the fact that we *have* a connection and that we can translate from runtime element to model element and vice versa. Some ideas for what we can use it for will be presented in section 9.2.1.

Chapter 5

Experiment: Use in Course

JFDebug was used by the students who participated in the course *INF5150 – Unassailable IT-systems* in the autumn 08 semester. In the course, the students modelled JavaFrame systems in Papyrus and transformed them to Java-code with JavaFrame Transformation. When they ran their systems, JFDebug would be used by default instead of regular JavaFrame. While the JFDebug features were made easily available, it was up to the students whether or not to use them.

The JFDebug *Eclipse plug-in* for Papyrus was not used. It only offered visualisation of information already available to in JFDebug (and not very much information). And while JFDebug would be activated by default, the plug-in would require the students to find the correct Eclipse view, select the right UML model and package and connect it to JFDebug. It was unlikely to get much use, or prove particularly useful to those who used it.

Some features have been added to JFDebug after the course. The version used in the course did not have the state machine windows of the current version. Double-clicking an entry in the state machines tab would open an event window for the most recent event for that state machine. The event window would not update as the state machine system kept executing transitions, or as the debugger was used to move back and forward in time. And it did not have the "Previous" and "Next" buttons for moving back or forward in time to the nearest event for that state machine. Also, there was no "Go to" functionality in the trace tab. Only the next and previous buttons in the main window were used to step in time. And the debugger could not export sequence diagrams.

In the last lecture of the semester, the students were given a questionnaire about their experience with the tool.

5.1 Goals

The systems created in the course were not very huge and they were unlikely to run for long enough for large numbers of events to occur. And test cases were not huge or overly complex. Performance issues related to scalability were very unlikely to arise during use, so the questions asked in the questionnaire did not deal with that.

What we hope to learn from the questionnaire results is if the features of the tool were of any use to the students. If they offered useful functionality and information, and if they were easy to use. Most of the questions are about whether or not the students had made use of the different features.

We also want to know if the students had used any other means of debugging than JFDebug, and if so, if that was because the JFDebug functionality was not sufficient for what they needed to do. As any Java code could be put in a transition and JFDebug is not too concerned with what goes on *within* each transition, there are many possible bugs that will require lower level debugging than what JFDebug offers.

5.2 Questions

The last question had a text field where the students could write any suggestions. For the other 6 questions, the students could answer yes, no, or N/A.

The questionnaire contained the following questions:

1. Have you used JFDebug?

   That is, had the student made use of any of the functionality or information offered by JFDebug? Since some students might not have participated much in all parts of the project work, or ignored JFDebug, it is useful to be able to filter out those who had not used JFDebug at all when looking at the results.

2. Have you used the functionality for stepping back in time for debugging/testing purposes?

3. Have you used the 'trace' tab for locating bugs or confirming that the program is behaving as intended?

   The trace tab offered similar information to that in JFTrace, which has been used in the course in previous semesters.

4. When you double-click an entry in the 'statemachines' or 'trace' tab, an event window with information about state and attributes pops up. Have you used event windows?

JFDebug did not have the state machine window functionality of the current version, and only opened event windows for entries in the state machines tab. Because of that, a distinction between windows opened from the trace tab and windows opened from the state machines tab was not very interesting.

5. Have you used any other means of debugging your generated code? (such as Eclipse's Java debugger or printing debugging information to console)

6. If you answered yes to the previous question, would it have been sufficient to only use JFDebug? (answer N/A if you don't know)

7. Any suggestions for improvement of JFDebug?

In previous semesters, JFTrace has been used to get a list of events similar to that in JFDebug's trace tab. So, question 2 and 4 deals with the major new functionality offered in JFDebug; if a student answered no to question 2 and 4, but yes to question 3, it is likely that JFTrace would have been as useful to him or her as JFDebug was.

5.3 Results

1	2	3	4	5	6
Yes	No	Yes	No	Yes	N/A
Yes	Yes	Yes	No	Yes	N/A
Yes	Yes	Yes	No	Yes	No
Yes	Yes	No	No	Yes	N/A
No	No	No	No	No	No
Yes	No	No	No	No	N/A
Yes	Yes	Yes	Yes	Yes	No
Yes	No	Yes	No	Yes	No
Yes	N/A	Yes	No	Yes	No
No	No	Yes	No	Yes	N/A
Yes	No	Yes	No	Yes	No
Yes	No	Yes	No	Yes	N/A
Yes	Yes	Yes	Yes	Yes	N/A
Yes	Yes	Yes	Yes	Yes	Yes
Yes	Yes	Yes	No	Yes	No

Table 5.1: Questionnaire results

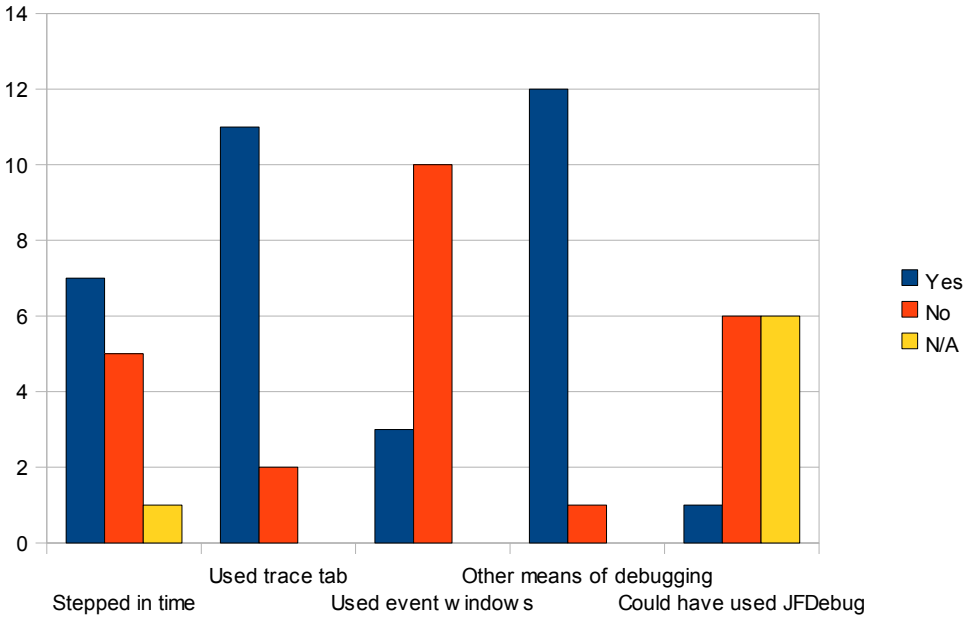


Figure 5.1: Results for the students who answered yes to "Have you used JFDebug?"

15 students attended the lecture and answered the questionnaire (while 18 students took the final exam of the course). Table 5.1 shows the questionnaire results.

Two of the students answered no to the first question, "Have you used JFDebug?". The second of these (10th entry in the table) answered yes to question 3 and 5. The answer to question 5 (used other means of debugging) makes sense. The answer to question 3 (used the trace tab) does not make sense: if the student did not use JFDebug at all, he or she could not have used JFDebug's trace tab. And so we can wonder if perhaps the student *did* use JFDebug, or maybe he or she did *not* use the trace tab. It is not obvious which one is the case. We are first and foremost interested in the answers from the students who did use JFDebug. Leaving the questionnaire answers as they are, the logical inconsistency in the tenth entry should not have much of an effect on the results we are most interested in.

Figure 5.1 shows a chart of the answers to questions 2–6 for the 13 students who answered that they did use JFDebug (the ones who answered yes to question 1). We see that:

- The debugger has been of some use to most of the students. Most of the students made use of the trace tab and about half the students used the debugger to move in time.

- Few students used the event windows.

- Almost all the students used other means of debugging in addition to (or instead of) JFDebug.
- About half the students could not have used JFDebug for all their debugging. And about half of them did not know if they could have used JFDebug.

The students were not given a lot of instructions on how to use JFDebug. The debugger was started whenever they started execution of their state machine systems, and they were encouraged to use it, but it was up to them to make use of it (and there was no instruction manual provided). It is not unlikely that some students simply were not aware of everything the debugger could be used for and how; a substantial amount of students that did not know if the debugger could have done what they used some other means of debugging to do. It is also likely that when something went wrong, many students would turn to means of debugging that they were already familiar with rather than trying our debugger.

If students did not know how to open event windows, that would certainly explain why they did not use them. It is also a possibility that some students knew of the event windows, but did not find them useful or easy to use. If the latter is the case, we hope that the state machine windows added to the debugger after the course would have addressed this issue for some of the students. The state machine windows make it more convenient to pay attention to state machines of interest while moving in time.

We assume that when a lot of students used some functionality, it was not very hard to understand how it could be used or what it could be used for. As almost everyone used the trace tab, it is unlikely that it was difficult to grasp what that was about. That is not very surprising: the trace tab is hardly a unique JFDebug feature, and more or less the same functionality has been offered through JFTrace in earlier semesters. And as about half the students did use the back-in-time functionality, we assume that it was not *very* hard to understand that either. As we did ask if the trace and back-in-time functionality was used for debugging/testing purposes, we can assume that the students who answered yes to those questions had an understanding of what they were doing with that functionality (i.e. they did not answer yes just because they knew that there was a trace tab or were aware of the "Previous" and "Next" buttons).

Only one student answered anything for the final question, "Any suggestions for improvement of JFDebug?" The suggestions made were:

- A more intuitive GUI.
  - A graphical representation of state.
- A user manual

- A visual state machine debugger.

It is not very surprising that a user manual would have been useful. The points about visualization (of states and state machines) may be addressed by integrating the debugger with the modelling tool. We want to take advantage of the diagrams made by the modeller instead of making new visualizations with the debugger (more on this in section 9.2.1). Currently, our Eclipse plug-in (that was not used in the course) does offer a graphical representation of state, by highlighting states and transitions in state machine diagrams.

Chapter 6

Overhead and Scalability Tests

6.1 Test Systems

In order to measure overhead caused by JFDebug, we have made 3 test systems. The two sources of overhead we are most interested in are the recording of events, and the creation of new state machines. Two test systems are made to test the overhead from recording events, and one for the overhead from creating new state machines.

The test systems are modelled in Papyrus and transformed to Java code with JavaFrame Transformation. Some of the behaviour of the state machines depend on whether we are measuring CPU requirements or memory requirements. These differences are explained in section 6.2.

6.1.1 Test System 1

In the first test system (figure 6.1), signals are sent between two simple state state machines. State machine A counts the number of signals sent and received (i.e. the number of events occurred). It calls `System.exit()` to stop execution of the state machine system once the count reaches a certain number. Both machines respond to incoming signals by sending new signals back, and state machine A sends one signal to B in its start transition. State machine A has two properties, *max* and *count*, in order to count events. State machine B has no properties.

6.1.2 Test System 2

In the second test system, signals are sent back and forth between state machine A and B the same way as in the first test system. The state machines have more properties in this system, so it should take longer time and require more memory to copy state machine information between events for this system.

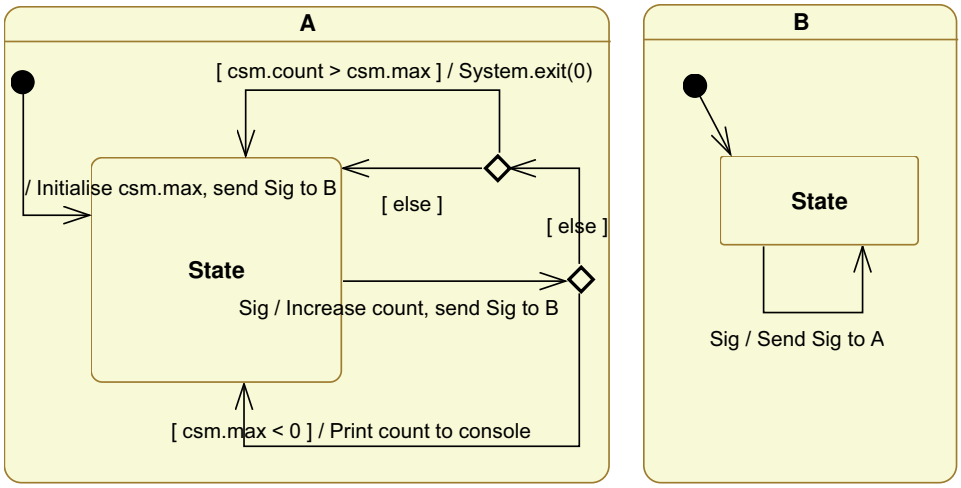


Figure 6.1: State machine diagrams for test system 1

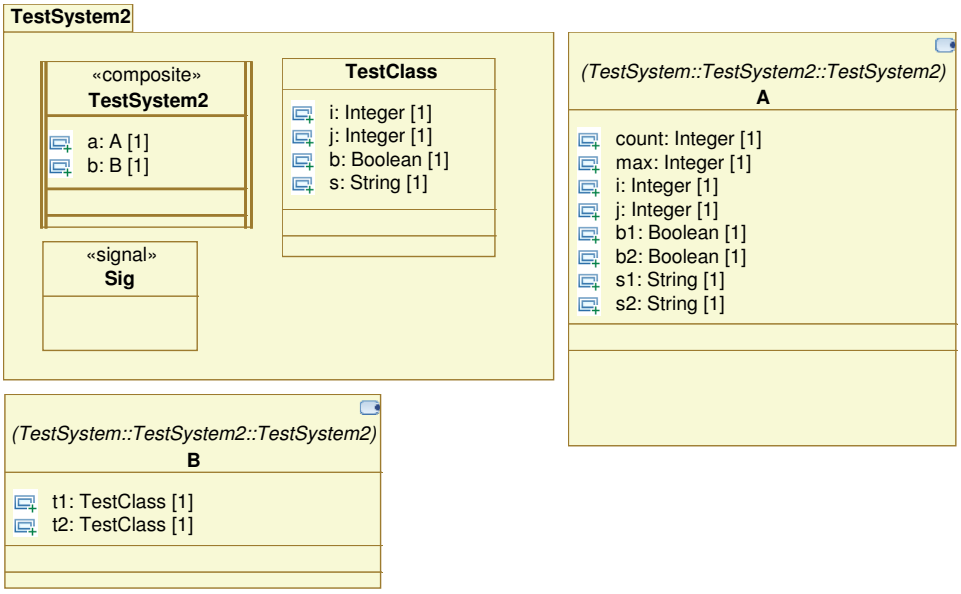


Figure 6.2: Class diagram for test system 2

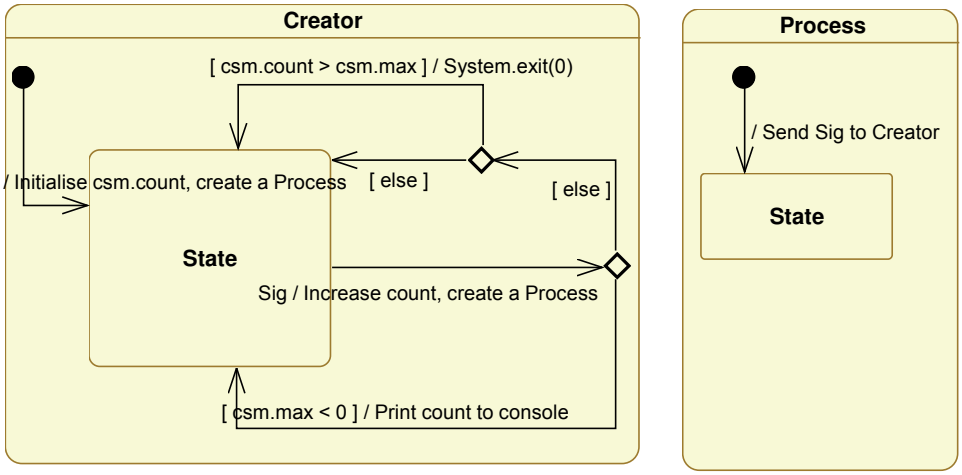


Figure 6.3: State machine diagrams for test system 3

Figure 6.2 shows test system 2's class diagram. State machine *A* only has primitive Java types and `String` properties. The debugger will copy the values of the primitive types, and because Java strings are immutable, it will only copy the pointer value for the those. State machine *B* contains classes that are "unknown" to the debugger, and the debugger will use the Java Serialization API for those (write the objects to byte arrays, and then read copies of them from those byte arrays). It is likely that *B*'s events take longer time to create, and require more memory, than *A*'s events.

This system's state machine behaviour is mostly as in test system 1 (figure 6.1), only with some additional Java code to handle the new state machine properties. State machine *B* initializes the TestClass objects in the transition from the initial state; the debugger does not check if any changes are made to the objects and will make new copies for every event. State machine *A* creates one new string for its *s1* property at every transition triggered by *Sig*; old strings can be garbage collected when using regular JavaFrame, but must are kept in memory when using JFDebug.

6.1.3 Test System 3

Creation of state machines during runtime causes additional overhead. New `StateMachineData` objects must be created, and the debugger must find all connections between the new state machine's mediators and other mediators.

In the third test system (figure 6.3), the *Creator* state machine will create a large number of *Process* state machines before shutting down by calling `System.exit()`. Created *Process* machines will send signals back to the *Creator* machine, and the *Creator* will respond to incoming signals by creating new *Process* machines. The *Creator* machine will create a *Process* machine in its start transition, and it will count the number of occurred events.

Typically, when creating state machines, a pointer to one of the state machine's mediators is going to be put in a list in another mediator, like a `MultiCastMediator` or a `SimpleIdRouterMediator`. As more state machines are created, more entries are put in the lists and it will take longer time to find all pointers to the mediators of newly created state machines. In this test system, we have no mediator used for the *Creator* to send signals to the *Process*, and so we do not attempt to measure that overhead. The reason for this is that JavaFrame uses a very similar routine for removing pointer to mediators upon termination of state machines, also taking longer time as more entries are in the different lists. If JFDebug has scalability issues related to state machine *creation*, then regular JavaFrame is very likely to have the equivalent issues related to state machine *termination*. And if we then address these issues in JavaFrame (for example by making a router mediator with a hash map instead of a linked list) then JFDebug can take advantage of that too.

6.2 Tests

Each system is run through a number of tests in order to measure CPU and memory usage. The different tests use regular JavaFrame, and 4 versions of JFDebug. A lot of time is spent on the GUI side of JFDebug when the debugged system is running. Event views (Swing containers) are created and garbage collected. Rows are added to tables, in the trace tab as new events occur and in the state machines tab as new state machines are created. It would be possible to make the GUI query the debug controller once the debugged system was paused instead of having it be updated as the system was running. Therefore, a "no GUI" version of JFDebug is used in some of the tests. In the no GUI version, the debug controller behaves exactly like in the regular JFDebug, i.e. it still calls methods to update the GUI, but all the GUI methods are replaced with dummy methods that do nothing. In addition, there are "instrumented" versions of regular and no GUI JFDebug. These instrumented versions time the execution of the JFDebug specific `startEvent` and `stopEvent` method calls.

6.2.1 CPU Usage

When measuring CPU usage, each test system runs for a set number of events before shutting down. The number of events is set in an environment variable, MAXEVENTS. Every test system is tested with several different numbers of events.

For each test system, the following tests are run:

1. Time the execution of the program when using regular JavaFrame.

2. JFDebug tests:
    a Time the execution of the program when using JFDebug.
    b Run the system with the *instrumented* JFDebug to time the execution of all the `startEvent` and `stopEvent` method calls.
3. *No GUI* JFDebug tests:
    a Time the execution of the program when using the *no GUI* JFDebug.
    b Run the system with the *instrumented no GUI* JFDebug to time the execution of all the `startEvent` and `stopEvent` method calls.

6.2.2 Memory Usage

By setting MAXEVENTS to less than 0, the test systems will not stop running after reaching some set number of events. Instead they will frequently print the number of occurred events to the console, so that it is possible to tell how many events occurred before running out of memory.

For each test system, the following tests are run:

4. Count the number of events occurred before the program runs out of heap space, using regular JavaFrame.
5. Count the number of events occurred before the program runs out of heap space, using JFDebug.
6. Count the number of events occurred before the program runs out of heap space, using *no GUI* JFDebug.

When using regular JavaFrame, the heap space required does not depend on the number of events that has occurred, and test system 1 and 2 require very little heap space. In test system 3, since new state machines are created during runtime, the program will eventually run out of heap space even when using regular JavaFrame, so only that system will use test 3.

By running Java with the command-line option **-Xmx***n* [1], the maximum amount of heap space can be configured. Tests 4.b and 5 are run several times, with different amounts of heap space available.

6.3 Results

6.3.1 Processor Usage

We should note that the transitions used in the test systems are "minimal". They provide a little test functionality (event counting) and make sure that

[1] `http://java.sun.com/javase/6/docs/technotes/tools/windows/java.html`

the debugger has something to do (events to create, sent signals to record, ...), but do nothing more. Our overhead is determined by the number of events, the data we must copy, signals sent, and state machines created and terminated. A lot of what a state machine system would typically do during a transition does not effect this overhead. E.g. if calculations made during a transition takes longer time, that means it takes longer time for the transition to execute, but not that the JFDebug overhead will increase. For this reason, an overhead *factor* is less interesting than the actual overhead per transition, per property, per signal sent, etc. In particular, we are interested in how many transitions are handled by our debugger per second.

Of course, the number of events per second depends on the computer running the systems as well. The tests have been run on an Acer Aspire 5720ZG:

- Intel Pentium Dual-Core T2310 @ 1,46 GHz
- 2 GB DDR2 SDRAM
- Windows Vista Home Premium (32-bit)

I.e. we are not using a very high-end computer; depending on the computer used, we may able to record events faster.

Events	1	10	100	1 000	10 000	100 000
JF	0,23	0,23	0,23	0,23	0,23	0,36
JF 1GB	0,54	0,28	0,28	0,28	0,28	0,4
JFD	2,89	1,4	1,78	3,09	11,95	95,04
JFD 1GB	2,33	1,64	2,01	3,32	13,99	104,33
JFDNG	0,52	0,25	0,25	0,31	0,31	0,67
JFDNG 1GB	0,37	0,36	0,37	0,37	0,42	0,81
Events	200 000	300 000	400 000	500 000		
JF	0,42	0,48	0,54	0,61		
JF 1GB	0,47	0,53	0,65	0,72		
JFDNG 1GB	1,42	1,75	2,33	2,61		

JF: JavaFrame. JFD: JFDebug. JFDNG: JFDebug with no GUI. 1GB means that the Java program was started with 1GB heap space.

Table 6.1: Test system 1. Test 1, 2.a and 3.a

Tables 6.1, 6.2 and 6.3 show the time spent, in seconds, to execute different numbers of events for each each of the test systems. The test runs have been timed using the Python timeit module[2]. *JF* is JavaFrame (test 1), *JFD* is JFDebug (test 2.a) and JFDNG is JFDebug with no GUI (test 3.a). *1GB* means that the test was run with the **-Xmx***n* option to set max

[2] `http://docs.python.org/library/timeit.html`

Events	1	10	100	1 000	10 000	100 000
JF	0,5	0,25	0,25	0,3	0,42	0,8
JF 1GB	0,36	0,28	0,28	0,34	0,47	0,9
JFD	1,53	1,4	1,78	3,4	14,13	109,67
JFD 1GB	1,88	1,7	2,07	3,63	13,74	108,91
JFDNG	0,28	0,23	0,3	0,48	1,11	6,1
JFDNG 1GB	0,35	0,34	0,4	0,59	1,15	6,23
Events	200 000	300 000	400 000	500 000		
JF	1,29	1,86	2,29	2,73		
JF 1GB	1,4	1,84	2,28	2,78		
JFDNG 1GB	12,08	16,82	22,89	27,26		

Table 6.2: Test system 2. Test 1, 2.a and 3.a

Events	1	10	100	1 000	10 000	100 000
JF	0,41	0,23	0,23	0,23	0,23	0,42
JF 1GB	0,27	0,28	0,28	0,28	0,34	0,47
JFD	1,41	1,4	1,78	3,15	12,14	103,33
JFD 1GB	1,64	1,64	2,01	3,39	12,25	96,93
JFDNG	0,23	0,23	0,23	0,3	0,36	1,3
JFDNG 1GB	0,29	0,29	0,31	0,34	0,47	1,34
Events	200 000	300 000	400 000	500 000		
JF	0,67	0,86	1,23	1,36		
JF 1GB	0,78	0,91	1,28	1,4		
JFDNG 1GB	2,23	3,34	4,79	5,4		

Table 6.3: Test system 3. Test 1, 2.a and 3.a

heap size to 1GB. Others are run with default settings. 1GB heap space is used to ensure that the tests do not run out of memory, or get very close to running out of memory. Every test is run with both heap space settings for up to and including one hundred thousand events. Test runs for more than one hundred thousand events are only run with regular JavaFrame (with and without 1GB heap space) and the no GUI version of JFDebug (only with heap space set to 1GB).

It is apparent that external factors have played some part in the results. Running a test several times does not always take the same amount of time, and a lot of the tests took longer when ran for one event than they did when ran for ten. However, the results are good enough to get a general idea of the overhead, at least as the number of events is increased. Where possible we use results from the tests ran for one hundred thousand events and more for making calculations. E.g. in order to get the time spent per event for one test, we take the result from the five hundred thousand events run, subtract the result from the one hundred events run, and then divide that by four hundred thousand. That way we get an average based on a large number of events, and we are ignoring the time used to start the system.

In all the tests using JFDebug, roughly one thousand events are executed per second. The overhead caused by JFDebug is substantial: in test system 2, the "slowest" one, a hundred thousand events are executed in about half a second when using regular JavaFrame. Comparing the JFDebug results to those where no GUI is used, we see that most of the overhead is related to updating the GUI, and not to creating and storing events. By modifying or replacing the GUI so that it is not updated during execution of the state machine system it should be possible to get results close to the "no GUI" results here.

The instrumented versions of JFDebug keep track of three things:

- t1: the total amount of time used by event number 1, 3, 5, ...
- t2: the total amount of time used by event number 2, 4, 6, ...
- total: the total amount of time used by all occurred events.

The time used is measured using `System.nanoTime()`. *t1*, *t2* and *total* is printed to console once for every 100 occurred events. Each test system runs for a hundred thousand events before shutting down.

The state machines/state machine types take turns executing transitions: *t1* will be the amount of time used by JFDebug for one and *t2* is the time used for the other. In test system 1 and 2, A will first execute its initial transition (t1), sending a signal to B, then B will execute its initial transition (t2), and then execute the transition triggered by the signal sent from A (t1). From here on, all *t1* events are B's transitions and all *t2* events are A's transitions. The first two transitions add to the "wrong" totals, but at a

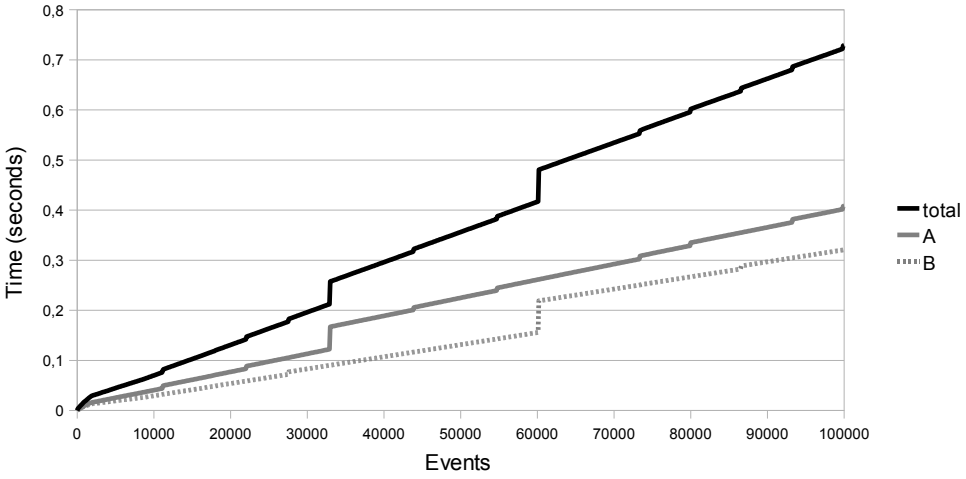


Figure 6.4: Test system 1. Test 3.a

hundred thousand events this should not make much of a difference. We say that *t1* is the total amount of time used by JFDebug for *B*'s transitions and *t2* is the total for *A*'s transitions.

For test system 3, *Creator*'s initial event (t1) creates a *Process*, then the new *Process* machine's initial event (t2) sends a new *Sig* signal to *Creator*. The signal triggers a new transition (t1) for *Creator*, which then creates a new *Process*, and so on. *t1* is the total amount of time used by JFDebug for *Creator*'s transitions and *t2* is the total for all the *Process* machines' transitions; no transitions add to the "wrong" total.

	Test system 1	Test system 2	Test system 3
t1	47,69	57,72	53,23
t2	47,74	50,3	49,36
total	95,42	108,02	102,58

Table 6.4: Test system 1, 2 and 3. Test 2.b

	Test system 1	Test system 2	Test system 3
t1	0,32	5,06	1,04
t2	0,41	0,44	0,25
total	0,73	5,5	1,29

Table 6.5: Test system 1, 2 and 3. Test 3.b

Tables 6.4 and 6.5 show the amount of time spent, in seconds, after a hundred thousand occurred events. These results appear to be consistent with those we got in test 2.a and 3.a (the *JFD* and *JFDNG* rows, respectively, in tables 6.1, 6.2 and 6.3). As we have seen, the overhead caused by creating

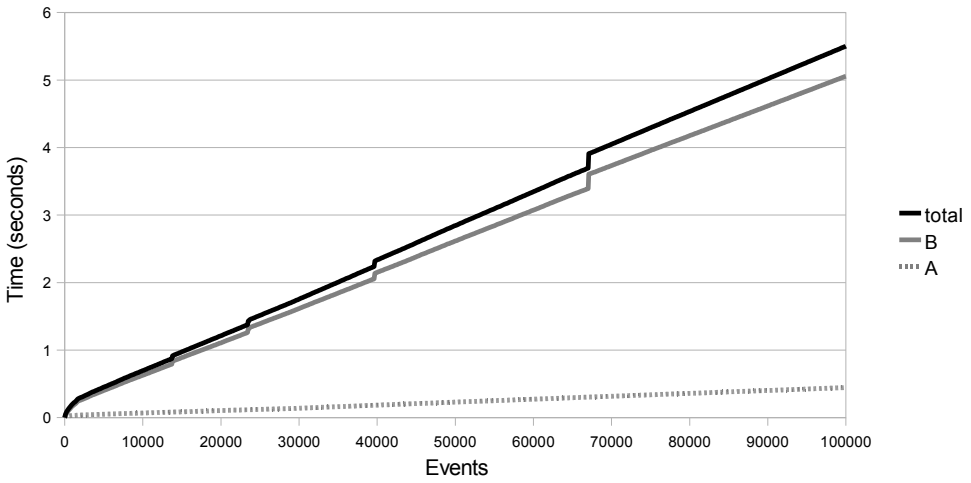


Figure 6.5: Test system 2. Test 3.a

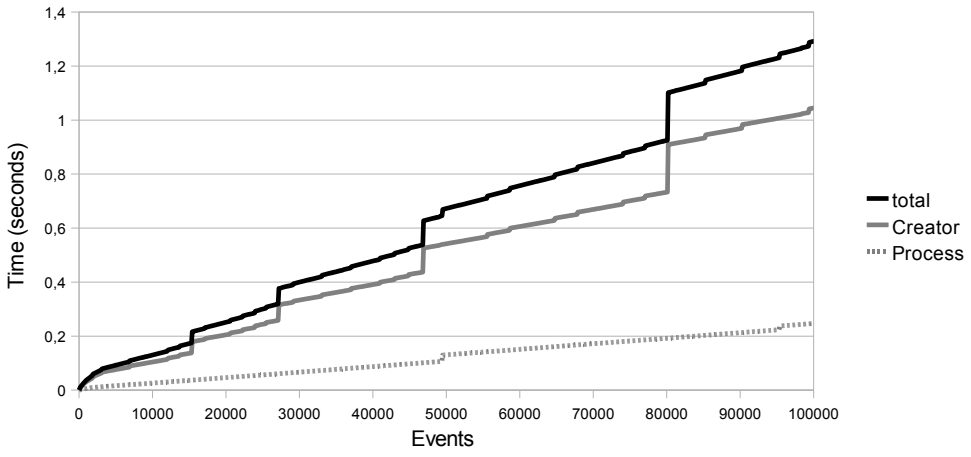


Figure 6.6: Test system 3. Test 3.a

and storing full states is overshadowed by what is caused by the GUI being updated. The differences between *t1* events and *t2* events are clearer in the 3.a (no GUI) results than in the 2.a ones. Graphs for the 3.a test runs can be seen in figures 6.4, 6.5 and 6.6.

With the instrumented versions of JFDebug, we have only timed JFDebug functionality *between* transitions. We know that there is a little overhead associated with sending signals to state machines in the debugged system (debug mailboxes report to the debug controller) and creating state machines (debug scheduler reports to the debug controller) as well. What we have timed with the instrumented JFDebug versions is roughly equal to the difference between using regular JavaFrame and no GUI JFDebug at one hundred thousand events. Meaning that the overhead we have *not* timed is not making much of a difference; as expected, the transitions are executing at close to full speed.

The results for the 3.a (no GUI) tests show how it takes different amounts of time to create events for the different state machines. In test system 1 (figure 6.4), it takes a little longer to create events for A than for B. This makes sense, as A has two properties, *count* and *max* that the debugger must copy between transitions, while B has no properties.

In test system 2 (figure 6.5), more time is spent creating events for B than for A. When creating events for B, the debugger uses Java's serialization API, and it is not surprising that this takes more time. Seeing how huge the difference is, it is worth looking into replacing the use of the serialization API with something else. At least when dealing with these fairly simple classes that are generated by the model transformation.

In test system 3 (figure 6.6), creating events for *Creator* takes more time than for the *Process* instances. The *Creator* has *count* and *max* properties, same as A in test system 1. The difference between the time spent creating events in A in figure 6.4 and *Creator* in figure 6.6 is the difference between sending one signal and creating one new state machine.

We also see that the graphs makes some quite large jumps in places, for example at a little more than 30 000 events and at about 60 000 events in figure 6.4. These are, presumably related to memory allocation and/or (attempts at) garbage collection; there are only two such jumps for test system 1, which requires the least memory, while there are more for test system 2 (which requires more space for copying full states) and test system 3 (which requires more space for creating `StateMachineMetaData` objects). The distribution of these jumps does not appear to be very "unfair": they have most effect on the transitions for the state machines that require most memory for their transitions. The *Process* of test system 3 has no properties, same as B in test system 1, and the transitions they execute are equally simple (both just send one *Sig* signal). Yet creating the events for B takes 0,32 seconds, while it takes 0,25 seconds for the events of *Process*. This appears to be due to the jump at about 60 000 events in test system 1.

If that had happened while creating an event for A, the results for B and *Process* would be closer to each other.

6.3.2 Memory Usage

Heap space (MB)	8	16	32	64	128
JFDebug	20 336	42 190	82 840	169 762	343 314
JFDebug No GUI	58 188	118 106	237 006	474 808	950 412

Table 6.6: Test system 1. Test 5 and 6

Heap space (MB)	8	16	32	64	128
JFDebug	12 378	28 098	59 248	124 142	246 782
No GUI	27 748	56 482	113 482	227 498	455 526

Table 6.7: Test system 2. Test 5 and 6

Heap space (MB)	8	16	32	64	128
JavaFrame	113 254	229 762	460 958	923 352	1 848 138
JFDebug	10 660	24 148	52 554	103 872	210 030
No GUI	29 050	58 968	118 330	237 056	474 504

Table 6.8: Test system 3. Test 4, 5 and 6

Tables 6.6, 6.7 and 6.8 show the number of events occurred before running out of memory for different heap space sizes. Figure 6.7 shows a graph of the results for all the tests (the numbers in the labels are test system numbers). The numbers of events occurred are always multiples of two, because that is how exact our event counting is. Only one state machine counts events, and it will increase the count by 2 each time (one for itself and one for the transition the other state machine must have made). As we can see, the numbers scale linearly; if we double the heap size we more or less double the number of events that can be stored. And again, we see that the results improve when we use JFDebug without the GUI, though the difference is not as huge as in the timed tests.

The number of full states stored is equal to the number of events occurred plus one for every state machine in the system (for each state machine the full state is stored after each of its events, and once before its first event). So in test system 1 and 2, the number of events is roughly equal to the number full states stored. In test system 3, the first event and every other event from there on creates a new state machine, making the number of full states stored about 1.5 times the number of events occurred. Also in test system 3, the number of `StateMachineMetaData` objects created and kept in memory

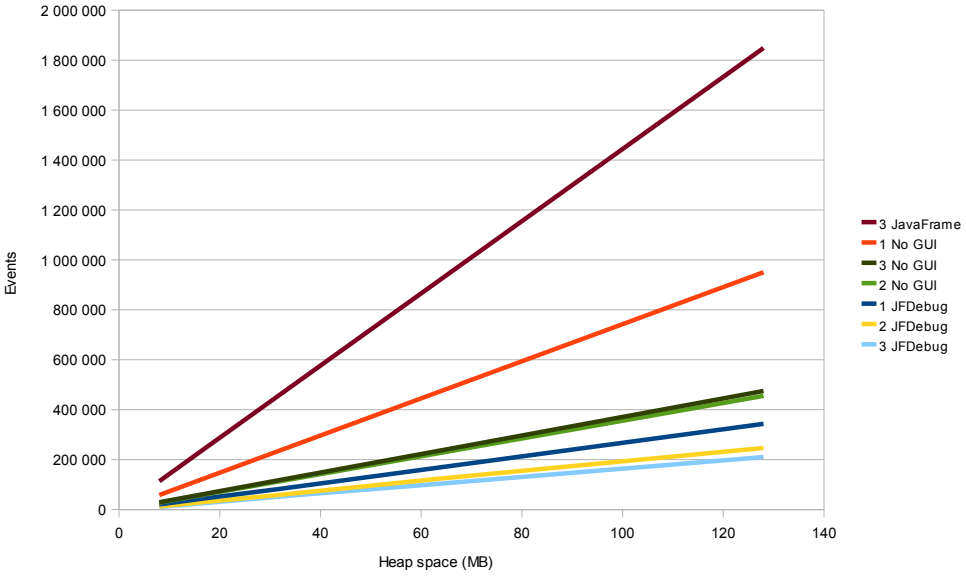


Figure 6.7: Test systems 1, 2 and 3. Tests 4, 5 and 6

is about 0.5 times the number of events occurred. In other words, some of the extra space required in the test system 3 tests is because more full state copies are made; not all of it is because of the `StateMachineData` objects (though as the *Procces* state machines have no properties, those "extra" full state copies are fairly small).

At 128MB, we don't use a very huge portion of the address space, and we have room for hundreds of thousands of events. How many events depends on the state machine systems. If we add more properties to the state machines, we require more space per event. As a careful estimate, we can say that if we keep the two hundred thousand most recent events in memory, we are unlikely to use more than 128MB of memory for storing events; at least if we do not use the current GUI.

Chapter 7

Related Work

In this chapter we will take a look at other back-in-time debuggers and approaches to back-in-time debugging. In section 7.5 we will discuss the differences and similarities between our approach and the others.

7.1 Omniscient Debugging

An omniscient debugger is a debugger that is able to store all events that change the state of the program under debugging, and use these stored events to show what the state of the program was at any point in the history of its execution. With a breakpoint debugger, the history of the the program execution is unknown when viewing the state of the program, and only what is currently in memory can be inspected. On the other hand, logging based approaches to debugging do store the history of the program execution, but leaves the user inspecting a potentially huge log/trace instead of the state of the program. Omniscient debuggers attempt to address these issues with traditional approaches to debugging. They are logging based, but the logged events have enough information so that the state of a program can be reverted to that of an earlier point in time. And then the state of that program can be presented to the user, instead of just the log.

Lieberman and Fry have proposed ZStep95 [19], a "reversible stepper" for Lisp. It stores the complete execution history of the program under debugging and allows the user to step back and forward in time, and offers graphical, animated views of the program under debugging. More recently, Bil Lewis' has created ODB [20, 21], an omniscient debugger for Java (section 7.1.1), and Hofer et al. have proposed Unstuck [22], a similar debugger for Squeak Smalltalk. Pothier et al. have proposed TOD [3], addressing scalability issues of previous back-in-time debuggers (section 7.1.2).

7.1.1 The Omniscient Debugger

Bil Lewis has created the Omniscient Debugger (ODB) [20], which is an omniscient debugger for Java programs. ODB records a collection of "time stamps" for a running Java program. With every time stamp, it stores information about the corresponding state change of the program. Time stamps are recorded for events such as assigning to variables, making method calls and throwing and catching exceptions. The bytecode for the debugged program is instrumented to make it record time stamps, and the recorded time stamps are kept in the debugged program's heap. After recording time stamps, the user is able to navigate through the history of the program execution, and view information about the Java program from any point in its history of execution.

In [21] Lewis presents some motivation for omniscient debugging. Often, a bug will have an effect that is easily discovered (like the program stopping and dumping a stack trace), while the cause of that effect is harder to locate. E.g. a null-pointer exception is caused by a variable being set to null, but it causes the program to crash at a much later point during execution. An omniscient debugger can answer the question of "Who set this variable?"

With a breakpoint debugger, the programmer must "guess" where to set breakpoints in order to catch the assignment that causes the crash; with an omniscient debugger the programmer can let the program crash and then step back in time to find the cause. When dealing with non-deterministic problems and using a breakpoint debugger, this becomes a lot more problematic. The programmer is trying to catch the cause of an effect that only sometimes occur. With an omniscient debugger, the problem has to be (re)produced in only one debug session.

When the program under debugging is paused or stopped, the ODB GUI (figure 7.1) can be used to navigate in time and get information about the state of the program under debugging. The GUI is one window separated into 8 panes. One is for threads; selecting a thread reverts the program back to the nearest time stamp for that thread. There are panes for the call stack, local variables, the `this` object, method trace and Java code, all showing information about the currently selected thread. There is also an objects pane, where the user can add any objects of interest in order to easily keep an eye on them while navigating in time, and a pane showing output from the debugged program to console. The user can then navigate in time by selecting or stepping through method trace lines, code lines, or output lines. He can also choose a variable of some object and move back or forward in time until its value changes.

Finally, there is a "minibuffer" (like in Emacs, at the bottom of the window) that can be used to execute Java methods. Methods executed this way get their own timeline and will not effect the execution of the program under debugging. Through the minibuffer, the user also has access to Ducassé's

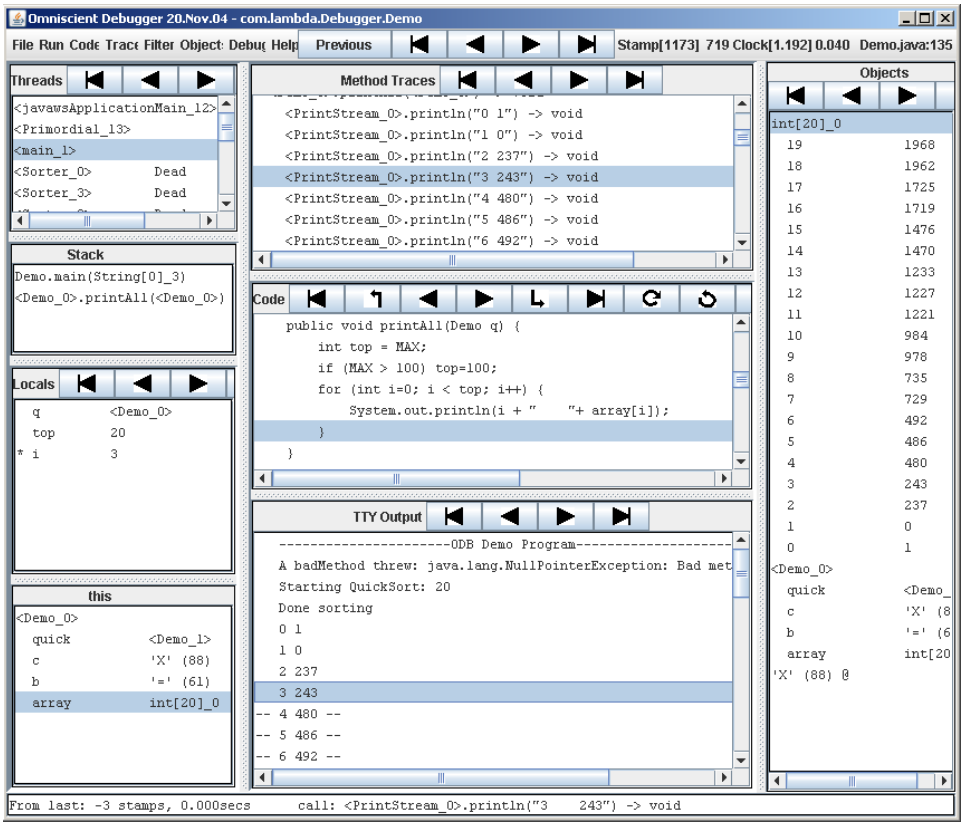


Figure 7.1: ODB GUI

`get` function [23]. The `get` function matches a pattern to an event. For example, the pattern:

```
port=enter & methodName="sort" & parmNames=["start", "end"]"
```

can be used to find the entry line for a method named "sort" with two parameters named "start" and "end". This can be used to search through the recorded events to find a point in time of interest.

7.1.2 Scalable Omniscient Debugging

Omniscient debugging has some scalability issues. While running a debugged program events must be recorded, which should be fast and non-intrusive. Storing and querying a potentially huge number of events requires space and time. In addition, it should be made as easy as possible for the user to find the information he is looking for when dealing with huge event traces.

The ODB does not deal with all of these issues. Storing time stamps is fairly intrusive; the code being debugged is instrumented to make it generate time stamps, and the time stamps are kept in the heap of the debugged program, using part of its address space. By Lewis' estimation, a 31 bit address space allows for about 10 million events. By throwing away older events, as long as the symptom and the cause of a bug are less than 10 million events apart, it should not be an issue. If that is not the case, the number of events recorded can be reduced by only instrumenting selected methods, and by starting and stopping recording at certain points in time.

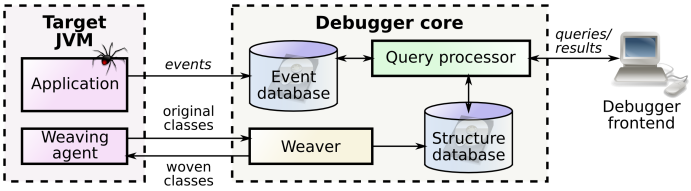
With ODB, recording can be started and stopped either manually (with a start/stop debugging button), or automatically. For automatic starting and stopping, Ducassé's `get` function can be used. In addition to requiring less space because fewer events are recorded, matching patterns to events tend to take shorter time than recording them. Given the pattern:

```
port=enter & methodName="sort" & parmNames=["start", "end"]"
```

(explained in the previous section), or one equally complex, one `get` test is about 40 times faster than recording one event. Problems associated with the execution overhead from using the debugger should be mitigated if only a small part of the execution needs to be recorded.

Other problems associated with huge event traces are not addressed with ODB. The issue with storing a huge event trace is dealt with by recording fewer events, making it a smaller event trace. Issues related to the user dealing with huge event traces or with querying huge event traces are not really encountered, as event traces larger than a certain size are not possible.

TOD (for Trace-Oriented Debugger) is an omniscient debugger for Java that addresses some of the issues that ODB does not deal with. In [3],



(figure from [3])

Figure 7.2: High-level architecture of TOD

(figure from [3])

Figure 7.3: Thread murals show the density of events in different threads

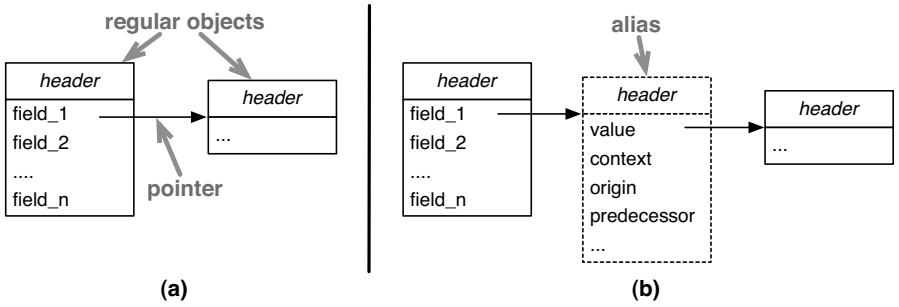
Pothier et al. presents the approach TOD takes to the scalability issues with omniscient debugging.

Figure 7.2 shows the high-level architecture of TOD. Like ODB, the bytecode of the debugged program is instrumented in order to record events. The weaver agent, a JVM TI [1] agent, is used to intercept class load events in the debugged program. The original Java classes are sent to the weaver, which instruments the classes and stores some information about them in the structure database. The woven/instrumented class is sent back and loaded in place of the original one in the debugged program.

Unlike in ODB, the instrumented classes only emit events; the events are not kept in the debugged program's heap. TOD uses a database cluster for storing events, with and a specialized database backend developed for storing events fast. The specialized database backend takes advantage of knowing that the events are *read only* and arrive ordered by timestamp, and that queries are limited to filtering. Both storing and querying events them scales well as the number of nodes increase. With the specialized database backend and 10 nodes in the cluster, up to five hundred thousand events are stored per second (and about 50 thousand with one node), while Postgres and Oracle supports storing 50 and 500 events per second respectively.

TOD can then store much larger event traces than ODB. In tests, TOD has stored about 720 million events, using 33GB of space. While ODB is limited by the amount of heap space available (i.e. once the debugged program requires enough heap space, the 10 million events estimate no longer holds), TOD is limited by the space available in the database cluster that has been set up.

[1] Java™ Virtual Machine Tool Interface, `http://java.sun.com/javase/6/docs/technotes/guides/jvmti/`



(figure from [24])

Figure 7.4: (a) Typical object format with references as direct pointers and (b) proposed extension

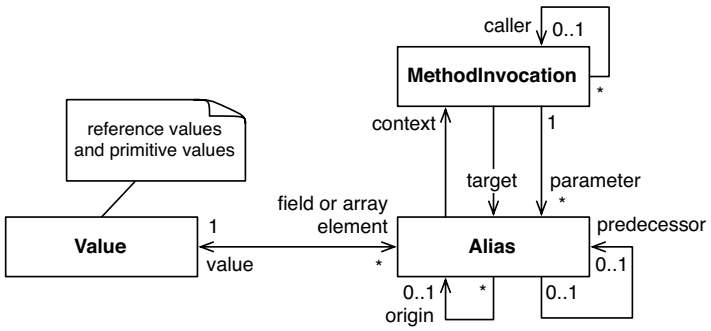
To make it easier for the user to deal with these huge event traces, TOD also offers higher level view of the execution history. *Murals* are used to show the density of events and method calls for the execution history of the program under debugging. Thread murals show the event density for different threads. An object mural shows the density of calls to methods of a given object, and a method mural shows the density of calls to a given method (on any object). Figure 7.3 shows one thread mural. The murals support zooming in and out, so that the user can get more detailed information about part of the execution history or an overview of all of it. When zoomed in, it is possible to select an event to move to its point in time.

7.2 Practical Object-Oriented Back-in-Time Debugging

In [24] Lienhard et al. present a different approach to back-in-time debugging. Instead of recording and storing all occurred events, their approach lets information about the history of the debugged program's execution be garbage collected as the program is running. By making information about the execution history of the program a part of the object model, at virtual machine level, they try to address memory/storage and performance issues with omniscient debugging.

They introduce the concept of *aliases* (figure 7.4). The aliases are invisible at application level: if the application is trying to read *field_1* in the figure, it will get the object pointed to by the alias *value* (or *value* keeps the value of a primitive type variable). Aliases are created whenever objects are allocated, passed as parameters, returned from methods, and when written to or read from arrays or fields.

The other fields of the aliases are for information about the history of the execution of the program for objects currently in existence. As seen



(figure from [24])

Figure 7.5: Conceptual object model with aliases capturing historical execution data

in figure 7.5: *context* fields point to the method invocation the alias was created during; *predecessor* fields are used for remembering the historical object state for the owner of the field the alias is for. *origin* fields are used for remembering the object flow [25,26] for the value of the field the alias is for.

The predecessor of an alias is an alias for the previous value for the same *field or array entry*, while the origin of an alias is a previous alias for the same *value* (for example, the origin of a alias for a value being read from a field is the alias for that value being written to the field). The *predecessor* can be used to find the previous values of fields, making it possible to reconstruct the state of an object at some earlier point in the program's history of execution. Only aliases for writing to fields and array entries have predecessors. The *origin* can be used to find out how some object was passed to the given field. The origin pointers can be followed back (in time) until the alias for the object's allocation.

An implementation of this has been done for the Squeak Smalltalk VM, and it would be a viable approach for other languages using virtual machines (like Java). Aliases are objects and will be garbage collected as they become unreachable from the current state of the program. This means that aliases are not discarded because they are old, but because they are no longer relevant, or at least less relevant, to the current state of the program under debugging. In a best case scenario, aliases are garbage collected so that the number of aliases will never go above some maximum number, and in the worst case the number still grows more slowly than with omniscient debugging. The trade-off is that it cannot be used to revert the state of the entire program under debugging to that of an earlier point in time: it is only guaranteed that the state of the objects currently in existence can be reverted.

7.3 Checkpoint-Based Debugging

Checkpoint-based debugging copies the state of the program under debugging at certain points in time, creating "checkpoints" that the state of the program can be restored from. We can consider our approach to back-in-time debugging could a combination of logging- or trace-based (like omniscient debuggers), and checkpoint-based: we use state machine-local checkpoints (and logging of certain events) to create events for our log.

An advantage of using checkpoints is that the code being debugged does not need to be modified. The challenge with using checkpoints is dealing with the potentially large amounts of data that must be copied from the running program under debugging. For example, IGOR [27] creates checkpoints by copying pages of memory from the running program to files. By only copying "dirty" pages, the amount of data that must be copied can be kept low compared to the data used in the program (depending on the spatial locality of the memory access pattern of the program under debugging). In *bidirectional debugger* (bdb) [28] copies of the state of the program under debugging is created by forking the process. This takes advantage of the operating system's write-on-copy policy, resulting in the same amount of memory being copied as in IGOR (though in this case the copying is handled by the operating system, during execution of the program).

For our approach, we know which state machine has executed a transition when we copy data, and we know that no other state machines have had their full state modified during that transition. Because of this, we know where to look for changed data, and we avoid the problems associated with more general purpose checkpoint systems (which is why we chose checkpoints for (part of) our approach).

7.4 Replay-Based Debugging

Replay-based debugging allows for moving back to a given point in time by replaying execution of the program up to that point. One challenge with replay-based debugging is non-deterministic execution. This can be addressed by logging any non-deterministic events. DejaVu [29], while not a back-in-time debugger (no moving back in time), offers deterministic replay by recording thread scheduling, user input and other non-deterministic events. The gain from using replay-based debugging is that all the execution that *is* deterministic can run at full or nearly full speed.

With replay-based debugging it can be a problem that a lot of execution is needed to get to the point in time we want, making stepping back in time slow. Bdb [28], mentioned in the previous section, combines replaying execution with creating checkpoints, so that instead of restarting the program, the state can be reverted to that of a checkpoint near the desired point in

time.

7.5 Comparison

With our approach, the debugger works on model-level. The other back-in-time debuggers typically work on (source) code-level. For example, the debuggers can step back and forward in time, one code statement at a time. Our approach does not let us do that; we cannot make steps in time that are shorter than one transition. With a code level back-in-time debugger, the user can find the location in the code where a variable had its value changed, while with our approach he can only find the transition where the value changed (and if it changed several times during one transition, our debugger does not know that). While we do not have this "low level" functionality, there are advantage to dealing with the debugged system on model level. In large and complex systems, presenting model-level information to the user should make it easier to get an overview of what is going on in the running system. As we create events less frequently, log of events that the user must navigate will be shorter, for a given amount of execution history.

Our approach is restricted to state machine systems; other debuggers typically work for any program written in a given programming language. Which means that other debuggers can be used for more different programs than ours. It also means that with our approach we know more about the programs that are being debugged, and can take advantage of what we know about state machine systems. E.g. because we know about transitions, we can take checkpoints between transitions, the user can easily move in time to the previous/next transition, and so on.

Like in omniscient debugging, we record each event with enough information to revert the debugged system to the state it was in before that event, although we store more information per event and create events less frequently. And like most of the omniscient debuggers, we present the state of the program under debugging at the point in time we have moved to. I.e. we navigate the history of execution and present the state of the program, and don't just present a trace of the execution history.

The way we use schedulers makes it possible to have parts of a state machine system in debug mode and others in production mode (we will discuss this further in section 8.3.3). Most other back-in-time debuggers instrument the code that is being run, and the user can select which classes and methods should be instrumented. With our approach, one state machine can be in debug mode while another runs normally, even if they are state machines of the same type. We note that Lieanhard et al.'s approach does not instrument code, and it might be possible to have some objects using *aliases* while others do not, even if they are objects of the same type.

As with other approaches that use checkpoints, an advantage of our ap-

proach is that we do no instrumentation of the code we are debugging. The code for the different transitions is the same whether our debugger is used or not. Most of the overhead caused by our debugger is between transitions instead of during transition (the overhead that is caused during transitions is when the "user-supplied" code calls certain methods offered by the framework). And because we do no instrumentation, the memory footprint added by our debugger is independent of the program under debugging. With ODB, for example, the classes of the program under debugging takes about three times as much memory after being instrumented. With our debugger, all "user-supplied" classes are the same as when not using our debugger; only the additions and modifications we have made to the state machine framework adds to the memory footprint.

Like in ODB and Lienhard et al.'s approach, our events are kept in the address space of the program under debugging. The amount of memory available for creating events depends on how much memory is required by the program we ar debugging.

Chapter 8

Discussion

8.1 Platform Requirements

The framework requirements for our approach are described in section 3.1; all the framework requirements are platform requirements. In addition to those there are some operations we need to do with our debugger that are unlikely to be needed for regular execution of a state machine system. As they are not required for regular execution, we do not expect framework support for them (although if we do have framework support, that is a preferred solution). In JFDebug we typically use Java's reflection API[1] for these operations. In the following sections we explain what those are, how we deal with them in JFDebug, and what we must be capable of doing if we use some other platform.

8.1.1 Copying State

For our approach, we must be able to copy the full state of a state machine. In the JFDebug implementation, we use Java's reflection support, as well as its serialization API.

For every state machine type, there is a `StateMachineClassData` object. These object contain Java `Field` arrays with one field for each UML property of that state machine. Before and after each transition we use `Field`'s `get` method to retrieve the values of the Java fields corresponding to the state machine's UML properties. The copied values are kept in `Object` arrays in the `Event` objects.

The most common values can easily be copied this way. For primitive values, and instances Java's wrapper classes for primitive types, we just copy the values from state machine to array. And for strings, we know that Java strings are immutable and we can just copy pointers to the strings. For instances of other, "unknown", classes we use the serialization API to write

[1] `http://java.sun.com/docs/books/tutorial/reflect/index.html`

the object, serialized, to some memory location, and then read it from there again. We do this to make sure that we don't just copy pointers that point to data that will be modified by the state machine in a later transition (i.e. to make sure that we get a deep copy).

After moving back in time, when resuming execution, we copy values back from `Object` arrays to the state machines using `Field`'s `set` method.

While we use reflection and serialization to copy state, we do not necessarily require support for that from our platform. What we do require is that we have some way of copying the full state of a state machine that ensures that no data is shared between the "copy" and the state machine. If we have a general purpose deep clone or serialization routine available, there should not be a problem. If we know that the data structures we are dealing with are very simple (e.g. state machines have only primitive type properties) we only need a shallow copy. If no appropriate copy method is available to us, it is likely that we can address it by making changes to the model transformation, for example generating a clone method for each state machine type.

8.1.2 Inspecting State

When we're inspecting the state machines in the system, we use the full state data copied when creating events. In JFDebug we use the data from the stored `Object` arrays to show property values and we use the field names from the `Field` arrays of the `StateMachineClassData` objects to show the names of the different properties.

While we could have used, for example, a C/C++ `memcpy` to *copy* the full state (if a shallow copy was good enough), we now need to access the individual properties of the state machine, and we need to know the name of those properties. We do need some basic reflection support for this (although for this too it is possible to build it into the model transformation if it is not a part of the programming language).

8.1.3 Adding and Removing State Machines

In JFDebug, we use reflection to put terminated state machines back in the state machine system, and remove created state machines from it, when moving back in time. In the `StateMachineClassData` objects we keep `Field` arrays with one field for each port/mediator of the state machine type. This is used to find other mediators that forward signals to a given state machine. This information is used to remove mediator connections when moving back in time to before a state machine was created, and to recreate mediator connections when moving back to before a state machine terminated.

The way we do this is the same way JavaFrame removes terminated state machines. While the code transformed from the models will create state

machines, the framework has a method for removing any state machine from the system. This method looks for mediator connections the same way we do it (so the framework requires reflection support whether our debugger is used or not).

If the framework we are using has some support for adding and removing state machines, we should be able to use that, but we might need to store additional information about connections between state machines (as we do for the mediator connections with JFDebug). If it is the code transformed from models that takes care of adding and removing state machines, and not framework code, it is very likely that we need reflection support in order to add and remove state machines.

In JFDebug, we also rely on Java's garbage collection here. We make sure that terminated state machines are still in memory by keeping pointers to them. In a language where memory is freed manually, we must either be able to stop the memory of a terminated state machine from being freed, or we must be able to recreate state machines when we need to add terminated ones back into the state machine system.

8.1.4 Summary

To summarize the previous sections, we require support for:

- Copying the full states of state machines in a way that ensures that no (mutable) data is shared between the copies and the state machines.
- Getting a list of properties for a given state machine type. And we must know the name of each property.
- Reading the value of any property, either from a state machine, or from a copy of a state machine's full state.
- Add/remove state machines to/from the state machine system. Other state machines in the system must "know" of state machines that we have added back to the system in the same way they know about state machines added when state machines are created during regular execution.
- Make sure that terminated state machines are kept in memory, or, alternatively, we must be able to recreate terminated state machine.

We do not expect the framework to offer support for all these things, as they are unlikely to be required in order to just run the kind of state machine systems we are interested in. And in order to do these things we need at least *some* reflection support, at least for getting property names and values. The reflection support can be provided by the programming language, or by an extension to the programming language, or it could be built into the model transformation that is used to generate code.

8.2 Usability

We have already done some evaluation of the debugger based on the questionnaire results (section 5.3). In (section 8.2.1) we will discuss the different ways the debugger can be used. In section 8.2.2 we will take a look at some challenging situations where it is difficult to determine what the debugger should do, and where resuming execution from earlier points in time may cause problems.

8.2.1 Usage Patterns

There are two different ways of using our debugger. The first is **history inspection**, i.e. to move back and forward in time, but only *view* information about the state machine system. The information about the execution history is used, for example to locate bugs, but it is not used to alter the state of the system (by resuming from an earlier point in time).

Another way of using the debugger is to move back and *resume execution* from an earlier point in time. We call this **reverting state**. This is particularly useful for testing. For example, if getting the system in a state where a bug can possibly occur takes time, we can reach it once and then just revert the state of the system back to it later. This can be useful when trying to reproduce a bug to locate it (for example because it occurred once when the system was not in debug mode). It can also be useful after locating a bug, to see if the bug still occurs when the system is given different input, etc.

Of course, these two ways can be combined. For example, after finding the cause of a bug by inspecting the execution history, the user might want to resume execution from right before that point and see if he can reproduce it. And viewing information about the system from an earlier point in time is always part of what is done before resuming execution from that point.

In addition to the different ways of using only our debugger, it is also possible to combine using our debugger with using another, code-level, debugger. We can use our debugger to find a transition with erroneous behaviour. Having isolated the bug to one transition, we can revert the state of the system to just before that transition, and then use a regular Java breakpoint debugger and step through execution of the transition. Of course, some advantages of using our debugger are lost if the other debugger we are using does not have the same advantages (e.g. if the other debugger instruments the code generated from the model transformation).

One of our issues with moving in time is that we risk leaving the state machine system in some state that could not have been reached during normal execution (an impossible state). We have addressed this by moving signals back in their queues when undoing the transitions they have triggered (section 3.4.1). While this has been a good enough solution for the systems we have used the debugger on, it will not always be good enough

(see for example *Event Effect* under section 8.2.2). It is also less likely to be good enough when doing *selective debugging* (section 8.3.3). So, depending on the state machine system, resuming execution from an earlier point in time is not "safe": leaving the state machine system in an impossible state can cause erroneous behaviour that could not have occurred under normal execution of the system. For larger state machine systems, it is important that the user of the debugger has an understanding of the risks involved in resuming execution from earlier points in time, which again demands an understanding of the running state machine system.

However, *inspecting history* is still completely safe. We can use the stored information about the execution history to locate bugs, without risking that we leave the system in an impossible state, as long as execution is not resumed from some earlier point in time. So even if it is deemed too risky to revert state, a lot of our debugger functionality can still be used.

8.2.2 Limitations

Sent Signals

We do not detect when a signal is sent from a state machine during a transition. We only detect it if it is put in the queue of another state machine in the debugged system. We do not need to detect signals sent out of the debugged system in order to move back in time: the signals sent out of the debugged system should not be "undone", as our debugger should not have effects outside of the debugged system. No erroneous behaviour is caused. However, what signals are sent during a transition is likely to be of interest to the user of the debugger.

In JFDebug, we detect signals when they are received by instances of the `DebuggerMailbox` class (*Signals* under section 4.4.2). If we want to detect messages sent from state machines in a similar manner, we need an modified version of the `Mediator` class. State machines in the debugged system could use debug mediators instead of regular mediators, and the debug mediators could report to the debug controller whenever they forwarded signals. Replacing mailboxes with debug mailboxes is fairly simple, as all the state machines use the same mailbox class. When dealing with mediators we have to take into account that some mediators might be instances of classes transformed from the UML model. We do not know about those classes until they are used, so we cannot have debug mailboxes ready for those in advance.

Because detecting sent signals have not been essential to moving back in time, we have decided to not deal with this issue, and only detect received signals. However we do know of two ways it is possible to deal with it. First, it would be possible to make the model transformation make debug subclasses of any mediator classes it generates. By naming the debug mediators appropriately, we can know that for any mediator class *Med* there exists a

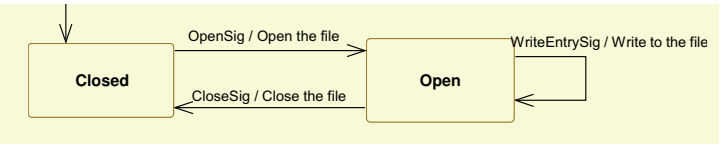


Figure 8.1: States/transitions for writing to a file

subclass *DebugMed* that we can load from the classpath. A second approach is to generate debugging subclasses during runtime. Javassist [30], for example, can be used to create debugging subclasses that reports to the debug controller whenever a method is called. With both of these approaches, we would have classes that only added overhead to the debugged system: the regular mediators would be used for any state machines not in debug mode (same as with our debug mailboxes).

Finally, this is a platform specific issue. Depending on the state machine framework used, sent signals may discovered just as easily whether they are sent to state machines in the debugged system or not. Either way it should be noted that we only need to detect signals being put in signal queues *within* the debugged system to be able to revert state, while information about signals sent *out of* the debugged system may be of interest to the user of the debugger.

Event Effects

If a transition causes something to happen, beyond modifying the state machine's full state, we want to record it and be able to "undo" it (*Event Effects* in section 3.2.1). We record state machines being created and terminated, and signals being sent to other state machines in the debugged system. We also record exceptions thrown during transitions, but we don't need to do anything special to undo those (reverting the full state to that from before the transition is enough).

The modeller of the system we are debugging is, in a sense, free to insert any Java code in order to do whatever he would like within transitions. We do expect some guidelines to be followed and will, for example, assume that mutable data structures are not shared between state machines. It still means that there are a lot of things that can possibly happen that we won't detect and record. These tend to be things that, by our definition, are outside of the debugged system (or the entire state machine system for that matter) and that it is not expected that the debugger should deal with. E.g., it should not be expected that SQL queries executed in some database will be "undone" by the debugger, or that text written to a file is removed by the debugger.

However, some times using something that is outside of the debugged system like this is reflected in the full state of a state machine, and as such

is part of expectations the modeller may be relying on when modelling the system. In figure 8.1 the modeller could normally expect that the file was open for writing when in the *Open* state. If, using JFDebug, he moves back time from when the state was *Closed*, to some earlier point in time where it was *Open*, that expectation will not hold. If he was relying on it, some erroneous behaviour is likely, possibly resulting in some exception being thrown.

We assume that the modeller of the state machine system has a reasonably good understanding of the Java code he has written himself. For state machine systems that are not very huge, it should not be too difficult to avoid such situations. And if it does cause some erroneous transition behaviour, we can usually just pause the system and step back in time again, this time making sure we step back until the state machine that caused the problem is in a safer state (e.g. *Closed*).

For larger systems, it might be harder to avoid such situations, particularly if the reason why we want to move to some point of time is unrelated to the state machine that causes the problem. Like if we want to get back to when one state machine was in a certain state, and some other then happens to be in a state that causes a problem. One solution to this, depending on the state machine system in question, might be to keep "unsafe" state machines out of the debugged systems. E.g. the *Archive* state machine that takes care of accessing the database and the *KML* state machine that handles writing of KML files (Google Earth/Maps files) can use the regular JavaFrame scheduler, while the rest of the state machines use the debug scheduler.

Timer Signals

JavaFrame has a signal type called *TimerMsg*. Timer signals are used by state machines to send signals to themselves after some amount of time has passed. A delay is set, a timer is started, and the signal will then be put in the state machine's mailbox after the delay has passed. The timer will run in its own thread once it has been started. In JFDebug we do not detect such timers being started, so we won't know about it until the timer signal is in the mailbox.

We do not give timer signals any special treatment. When a timer signal is put in a queue belonging to a state machine in the debugged system, we check the current thread and, seeing as it is not the debug scheduler's thread, we decide that it must an external signal. As with other external signals, we do not put it back in the queue when undoing a transition triggered by a timer signal. And as with other transitions triggered by external signals, if we did put it back in the queue, there would be no earlier transition where it would be natural to move it out of it again (because we don't know which transition started the timer).

We can solve this by making changes to the *TimerMsg* class. However,

this would change how it worked whether or not it was being used by state machines in the debugged system or by state machines using the regular JavaFrame scheduler. We prefer solutions where any overhead is added only to the debugged system. For now, the users of the debugger must be a little careful when dealing with timer signals. In practice this has not proved to be an issue.

A possible way of dealing with this issue, that *does* keep added overhead restricted to the debugged system, is to replace timer signals in the debugged system with instances of a subclass of *TimerMsg* (say, *DebugTimerMsg*) that does report to the debug controller when it is started. Timer signals are usually properties of the state machines they are sent to, so by finding *TimerMsg* properties and replacing them with *DebugTimerMsg* instances, we could make sure that we know it when timers are being started. By checking the properties of the state machines between all their transitions, this works as long as no timer signal is created and its timer is started during the same transition. The convention is that timer signal instances are created only when the state machines they belong to are created, so this should catch most cases of timers being started.

Also it is likely that, as we develop the debugger further, it will be natural to give the user more control over how external signals are treated when moving back in time (see section 8.3.3). This will let the users deal with issues related to state machines expecting external signals after moving back in time, whether those external signals are timer signals or not.

This is another platform specific issue. Other state machine frameworks are not unlikely to have some similar functionality for timing. But depending on the state machine framework used it might not be an issue, or it might be easier to deal with.

8.3 Scalability

Every event our debugger stores takes some space. The exact amount of memory depends on the state machine system we are dealing with. State machine properties, signals sent and state machines created and terminated all effect the required memory per event. No matter how much memory is taken by each event, if a system under debugging keeps running we will eventually run out of space. Whether the overhead in time is an issue also depends on the state machine system we are debugging. In particular it depends on how frequently the state machine system has to execute transitions.

We will discuss selection of state machines (section 8.3.2), and two ways it is possible to make the logged execution history require less space: by only recording *changes* in full states (section 8.3.1), and with *selective debugging* (section 8.3.3). Selective debugging can reduce the overhead in both time and space.

Of course, another way to address the issue of overhead in space is to make more space available, e.g. by writing events to disk/database. We will still assume that we do not have infinite space available, and that under some circumstances it is useful to reduce the amount of space required for logging as well.

8.3.1 Recording Changes

When we copy full states, it is possible to store property and state values only if they have changed during the transition. We can compare the new values to the ones from before the transition we are creating an event for, and only store the new ones. Depending on the state machine system, this may reduce the amount of memory required. However it might also increase it. Because we keep values for all the properties in each full state copy, we can store them as a list of property values; we know which property each value is for because the same index in each list of values is for the same property. If we only store values for some properties, different lists (for the same state machine) will have different numbers of entries. And we must keep track of which property each value is for. E.g. instead of having a list of `Object` instances, we would have one `Field` and one `Object` for each entry in a list. If, in every event, every property of the current state machine changed, this would require more space; if properties seldom changed, it could require less space.

8.3.2 Selecting State Machines

When dealing with large sets of state machines, a way of finding state machines and events of interest is useful. A way to deal with this is to let the user specify queries that state machines are checked against to make selections.

As a minimum, it should be possible to:

- Select state machine type.
- Select state (from the set of possible states for the chosen state machine type).
- Select values for the different properties.

For JFDebug, apart from listing state machine types and states, we are dealing with general purpose querying of sets of Java objects. While we can code our own support for (very) simple queries fairly easily, there are existing projects that we can use instead. For example, we can use JQL [31] or Quaere[2] to handle complicated queries and query optimisation.

[2] `http://xircles.codehaus.org/projects/quaere`

It will also be useful if the debugger remembers result sets from previous queries, and let the user perform set operations on these (union, intersection, complement, ...) to get the set of state machines he is interested in. As well as letting the user select individual state machines or groups of randomly selected state machines from reult sets.

8.3.3 Selective Debugging

We are able to move state machines between different schedulers. Currently we use this to turn debugging on and off for the entire state machine system. When debugging is turned off, all the state machines use the regular scheduler, and when debugging is turned on they use our debug scheduler. It would also be possible to use this to have some state machines using the debug scheduler while others are using the regular scheduler. We call this **selective debugging**. The state machines using the regular scheduler will suffer little overhead in time, and we will save space by only creating events for some of the state machines.

Assuming we have support for selecting state machines, as described in the previous section, sets of state machines could be selected and moved between schedulers. It would also be possible to have a third, "filtering" scheduler that checked state machines against queries and moved them to the debug scheduler when certain conditions where met. The filtering scheduler would add overhead to the execution of its set of state machines, but as it would not need to create events, space could be saved.

Reverting State

By putting internal signals back in the queues when undoing the transitions they have triggered (section 3.4.1) we have made it less likely that moving in time will put the system in some impossible state. If only parts of the state machine system are in the debugged system, this becomes harder to do. External signals can then be signals sent from other state machines in the system (just not in the debugged system), and it is (more) likely that some of those should be put back in their queues as well.

We could put signals back in the queues when undoing transitions triggered by signals sent from any state machines, and not just when the triggering signals where sent from other state machines in the debugged system. However we would have no earlier event for the transition that sent that signal, because that transition was for a state machine outside of the debugged system. Which would mean that we would have no earlier point in time where it would be natural to take that signal out of the queue again.

Even if we could easily solve the issue with external signals, reverting state is a lot riskier when only some of the state machines are in the debugged system. For example if the state machine in figure 8.1 on page 78 was outside

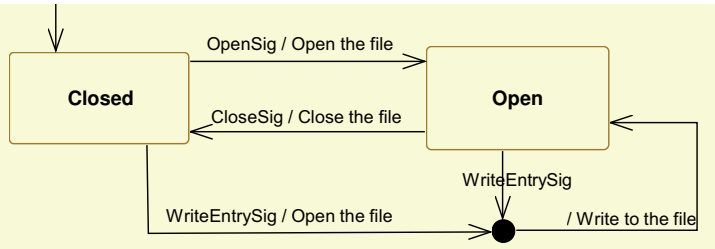


Figure 8.2: Revision of the state machine from figure 8.1

of the debugged system while a state machine that sent signals to it was not: if we reverted the state back to before a *CloseSig* signal was sent, the state machine in the figure would not be reverted back to its *Open* state. After resuming execution, the other state machine could send *WriteEntrySig* signals, expecting an *Open* state.

There is no way for our debugger to figure out what should be done in all such situations. And how big an issue this is depends on how robust each of the state machines are. If instead the state machine was like in figure 8.2, it might not be an issue, or at least less of an issue.

It would, in the end, be up to the user of the debugger what to do. Inspecting the history of execution would still be completely safe. How risky reverting state would depend on the state machine system being executed and what state machines were selected for debugging. It should also be up to the user what should be done with triggering signals when undoing transitions (though putting internal signals back in queues while throwing away external ones could still be the default behaviour).

8.4 The Event Model

In section 3.2.1 we described the event model we are using. Here, we will do the following:

- Section 8.4.1: Make a brief comparison between our event model and the events in sequence diagrams, and look at how we can group sequence diagram events into transitions or "debugger" events. This will be useful in the two following sections.

- Section 8.4.2: Introduce the concept of *signal chains*, and discuss how it can be used when considering throwing away events.

- Section 8.4.3: Discuss how we can apply the logic behind *weak sequencing* to undoing events with our debugger.

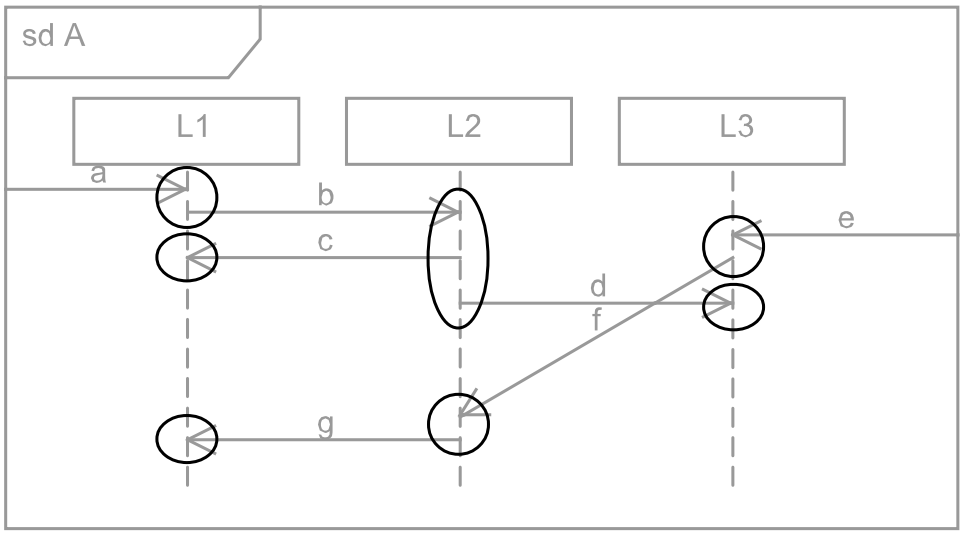


Figure 8.3: A sequence diagram with events from our event model marked with ellipses

8.4.1 Sequence Diagrams

In sequence diagrams, we have events for the sending and the reception of signals. For a signal s, $!s$ is the the event for the signal being sent and $?s$ is the event for the signal being received. In our event model, each event is for a transition, and each transition is triggered by the reception of a signal. We say that *dbg(s)* is the "debugger" event for the transition triggered by s.

While debugger every event has one triggering signal, several signals can have been sent during its transition. In figure 8.3 we have marked debugger events with ellipses. The set of *sent* and *received* events for *dbg(a)* is $\{?a, !b\}$. For *dbg(b)* it is $\{?b, !c, !d\}$, and so on.

Figure 8.4 shows the trace tab for the series of events that are exported to the sequence diagram in figure 8.5. We have marked the events in the sequence diagram with ellipses, as in figure 8.3, and labeled them with the event numbers for the corresponding events in the trace tab. As we see, no messages are sent from lifelines and to the edge of the interaction (the messages from the edge, to lifelines, are the external signal). As we said in section 8.2.2, we do not detect all sent messages, but only those that are received by state machines in the debugged system. So we do not always know about all *send* events that really occurred during a transition.

8.4.2 Signal Chains

In the approach to back-in-time debugging proposed by Lienhard et al. in [24] (section 7.2), they use the concepts of object state history and object

void | main | statemachines | trace

...	statemachine	signal	beforestate	afterstate
0	ICUsystem_Archive(0)	StartMessage()	(initial state)	Idle
1	ICUsystem_ICUcontroller(1)	StartMessage()	(initial state)	GeneratorState
2	ICUsystem_ICUcontroller(1)	* Sms(a a a hotpos, 203...	GeneratorState	GeneratorState
3	ICUsystem_ICUprocess(2)	StartMessage()	(initial state)	Idle
4	ICUsystem_ICUprocess(2)	Sms(a a a hotpos, 2034...	Idle	HotspotPosition
5	ICUsystem_ICUcontroller(1)	* PosResult()	GeneratorState	GeneratorState
6	ICUsystem_ICUprocess(2)	PosResult()	HotspotPosition	WaitNearestHot...
7	ICUsystem_Archive(0)	GetNearestHotspot(10.7...	Idle	Idle
8	ICUsystem_ICUcontroller(1)	NearestHotspot(Ifi, 1877...	GeneratorState	GeneratorState
9	ICUsystem_ICUprocess(2)	NearestHotspot(Ifi, 1877...	WaitNearestHotspot	FinalState

Figure 8.4: Trace tab for the sequence diagram shown in figure 8.5

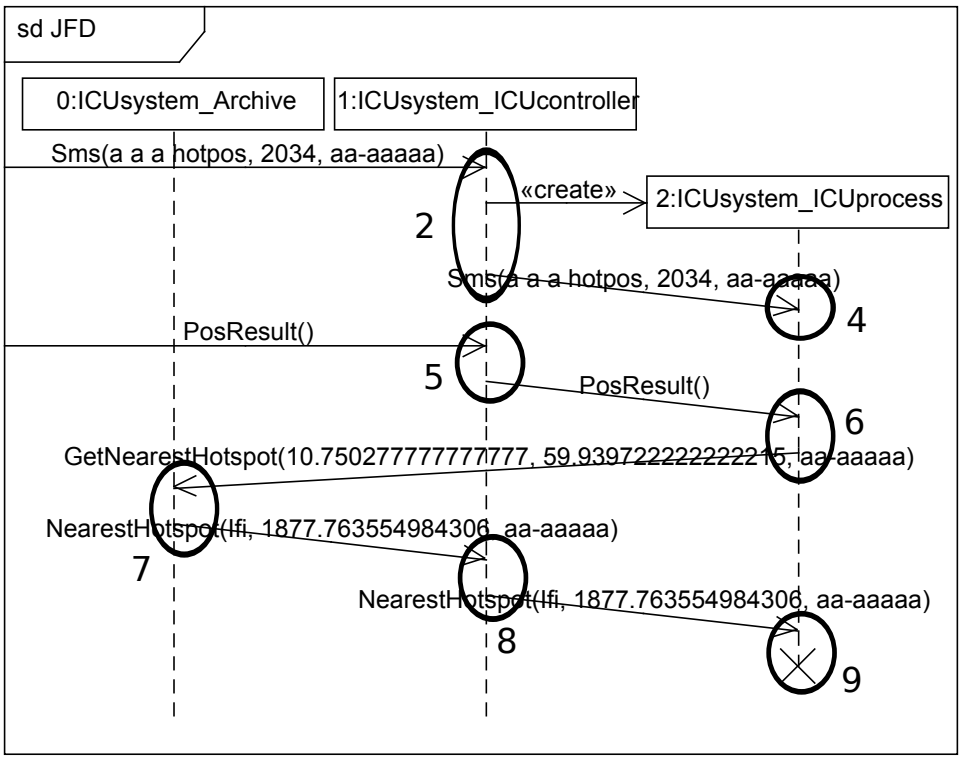


Figure 8.5: Sequence diagram exported from the trace shown in figure 8.4

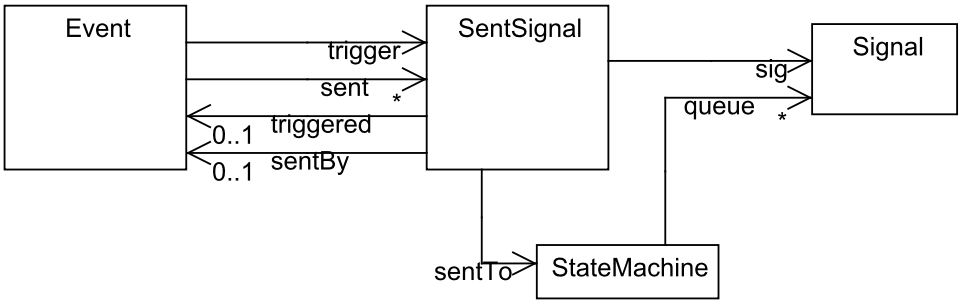


Figure 8.6: Signal chain extension to the event model

flow [25, 26] history.

In our approach, we have the state history of state machines as part of the event model. In our event model (figure 3.3 on page 17), we can navigate to a state machine's previous states using the event's *previous*. This is similar to how the *predecessor* of an *Alias* is used (figures 7.4 on page 68 and 7.5 on page 69). The *predecessor* is used to find previous values for fields while our *previous* is used to find previous states for state machines.

We do not have an equivalent of the *origin* of an alias. The *origin* is used to find out how the value of a field got there. Given an alias for a field, we can use the *origin* to trace our way back to the allocation alias for the object that is the *value* of the alias. This is similar to how we, given a transition, can trace our way back, following the transitions that sent the signal that triggered our current transition, until we get to an external signal or an initial transition of a state that was created when the state machine system started running (as we described in section 3.4.1).

We call sets of signal that are related to each other in this fashion for **signal chains**, or **event chains** ("chain" as in "chain reaction"; we do not call it a "flow" as one chain is not one signal "flowing" through the system, but is a set of signals where one signal caused the next to be sent). For example, in figure 8.3 we have the following signal chains:

1. $\langle a, b, c \rangle$
2. $\langle a, b, d \rangle$
3. $\langle e, f, g \rangle$

As it is, our debugger architecture does let us deal with signal chains. The event's *trigger* holds the signal that triggered the transition, and all the signals sent during a transition can be found by following its *sent*. Given an event, we can search through older events until we find the one that sent its *trigger*. For example, we do this when we export sequence diagrams with JFDebug (because we need to know the sender and receiver of a signal in

order to draw it). Making it part of the event model, as in figure 8.6, we can make it more straightforward. The event's *trigger* would then be a *SentSignal* instead of a *Signal*. We can follow an event's *trigger.sentBy* to find the event for the transition that sent the signal. And by following *sent.triggered* we find the events "caused" by the current event (it is to *trigger.sentBy* what *next* is to *previous*). Following *trigger.sentBy* from *dbg(c)* or *dbg(d)* gets us to *dbg(b)*. Following *sent.triggered*, from *dbg(e)* we get *dbg(f)*, and from *dbg(c)* we get both *dbg(b)* and *dbg(d)*. And so on.

We would then want to create *SentSignal* instances whenever signals were put in debug mailboxes (currently, there are *SentSignal* instances only for internal signals, as only those are sent from state machines in the debugged system). Once the signal of a *SentSignal* triggered a transition and we created an event, we would add that event to the *SentSignal*'s *triggered*.

With these changes, we would no longer have to search through the list of events in order to navigate back (from triggered transition to signal being sent) and forward (from sent signal to transition being triggered) in signal chains. More interesting, we can use this for choosing events to discard, in a way similar to how the garbage collection works in Lieanhard et al.'s approach.

Analogous to aliases that will not be garbage collected, we define **relevant events**:

- An event is relevant if it is the most recent event for a state machine (still) in the debugged system.
- An event is relevant if it is the *previous* event of a relevant event.
- An event is relevant if it is the *trigger.sentBy* of a relevant event.

And other events are, at least, less relevant (see figure 8.7). An event like that only changed the state of a state machine that is no longer in the system. And no subsequent events for that state machine sent any signals that changed the state of any *other* state machines in the debugged system. The event is not relevant to anything currently in the system. In figure 8.3, if *L1* had terminated, the *dbg(g)* and *dbg(c)* events would not be "irrelevant" events. If *L3* had terminated, *dbg(d)* would be be irrelevant. All the events would still be relevant if only *L2* had terminated.

If we remove the event's *next* and *triggered* from the event model (i.e. we would not be able to (easily) navigate forward in state history and signal chains) and remove the global list of occurred events, then, by removing terminated state machines, only relevant events would be reachable. Garbage collection could remove irrelevant events the same way it removes aliases in [24].

However, for our approach to debugging, we usually do want terminated state machines to be reachable. It is not uncommon for state machines to

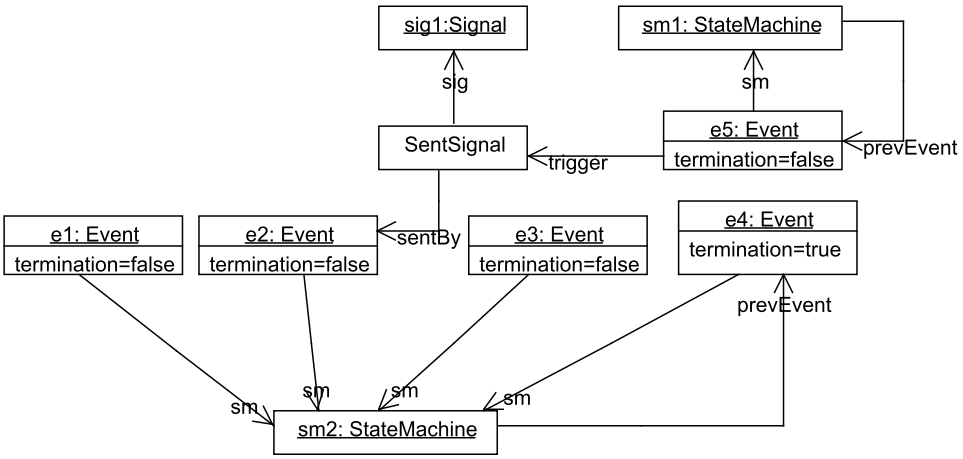


Figure 8.7: A sequence of events. *e1*, *e2* and *e5* are relevant events. *e3* and *e4* are not

send some signal *out* of the system in its final transition, like an SMS signal to a user of the system. Then, if we're getting incorrect output, the state machine that sent it will be unreachable by the time we discover that. Still, by defining relevant events as above we can choose to at least discard irrelevant events before relevant ones, if we do have to discard events. Depending on the state machine system being debugged, discarding the oldest of the irrelevant events first may prove valuable.

It would be possible to keep sets of relevant and irrelevant events up to date as the state machine system executed. However, this would require potentially large numbers of events to be checked each time a state machine terminated. Instead of causing more overhead during execution, we could check all the events in the system when appropriate (e.g. when the user decided, or when a certain amount of memory was used).

We could use the following routine for finding events to discard events:

1. Put every recorded event in an "irrelevant" pool.
2. For every state machine in the system, find its *prevEvent*.
3. Remove all the events we found from the "irrelevant" pool.
4. For each event we removed (every event that we found and that was not already removed from the pool), we find its *previous* event and its *triggeredBy.sentFrom* event.
5. Repeat step 3–5 for every event we found.

When no more events are found, we can throw away all events that are still in the "irrelevant" pool.

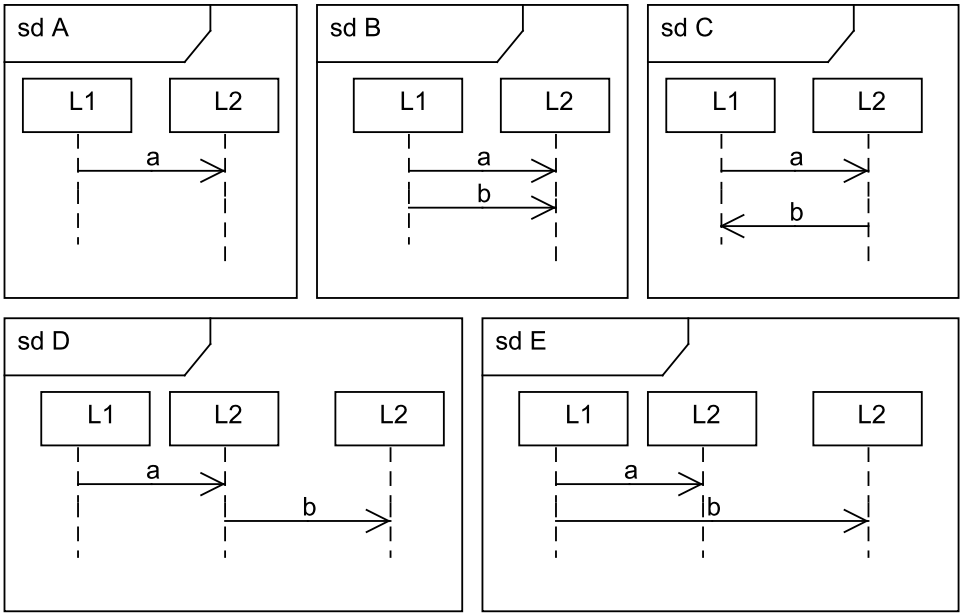


Figure 8.8: Sequence diagram examples

8.4.3 Weak Sequencing

In STAIRS [32,33], the weak sequencing operator *seq* is used to find sets of traces that describe interactions. A trace is a sequence of events, typically of signals being sent and received between lifelines.

We use figure 8.8 for examples. Given an interaction, by using the weak sequencing operator we get a set of all traces that meet the following criteria:

1. The order of events on each life line is maintained (the order is given by the vertical position on the lifelines). In B, the event $!a$ event must come before $!b$. In C, the event $?a$ must come before $!b$, etc.

2. For any signal s, the event $!s$ comes before $?s$. E.g. the only trace describing the semantics of A is $\langle !a, ?a \rangle$.

This is similar to how Lamport clocks [34] do partial ordering of events in distributed systems.

Weak sequencing takes into account that different lifelines are independent of each other. Unless point 1 above demands a certain order, events on different lifelines can occur in any order. For each of A, C and D, there is only one possible trace. In B, $?a$ and $!b$ can happen in any order. In E, $?a$ can occur before $!b$, between $!b$ and $?b$, or after $?b$. The sets of traces for the sequence diagrams in the example figure are:

- $A = \{\langle !a, ?a \rangle\}$

- $B = \{\langle !a, ?a, !b, ?b\rangle, \langle !a, !b, ?a, ?b\rangle\}$
- $C = \{\langle !a, ?a, !b, ?b\rangle\}$
- $D = \{\langle !a, ?a, !b, ?b\rangle\}$
- $E = \{\langle !a, ?a, !b, ?b\rangle, \langle !a, !b, ?a, ?b\rangle, \langle !a, !b, ?b, ?a\rangle\}$

This logic is applicable for the state machine systems we are dealing with. The modeller of a system cannot enforce a certain execution trace. For example, if state machine *SM1* sends *sig1* to *SM2* and *sig2* to *SM3*, the modeller must take into account that either *sig1* or *sig2* may trigger a transition before the other one.

Once those three transitions have occurred in the running system, we do know the order. Our debugger records the events for the transitions in the order they occur. And as of now, if we pause the state machine system and move back in time, we undo these events in the reverse of that order. By doing that, we are sure that the state machine system is put in a state that it was previously in. However, we could instead undo the events in any order we would like, as long as we did not break the weak sequencing of send and receive events. If we did this, we would be sure that we put the state machine in a state that it *could* have gotten to during normal execution (i.e. not an *impossible state*), just not necessarily one that it was in during this debugging session.

Using figure 8.3 as an example, we can undo *dbg(d)* first, or we can undo *dbg(g)*. Both *dbg(d)* and *dbg(g)* are the last events for their state machines (*L3* and *L1* respectively), and there are no signals sent that keeps us from "reordering" these events. We cannot undo *dbg(c)*, as *L1* has more events later in time. And we cannot undo *dbg(f)*, as that would be undoing the *send* event for g before undoing its *receive* event. If we start by undoing *dbg(d)*, our only choice then is to undo *dbg(g)* and then *dbg(f)*. If we have undone *dbg(d)*, *dbg(g)* and *dbg(f)*, we can then chose to undo either *dbg(e)* or *dbg(c)*, etc.

This relates to the *signal chain* concept we introduced in the previous section: we can undo any event that is the most recent of both a signal chain and a state machine. We recall that c, d and g were the last events of the three signal chains; either *dbg(d)* or *dbg(g)* can be undone first, but not *dbg(c)*, because *L1* has another event after *dbg(c)*.

As we said in section 3.4.1, we put internal signals back in the queues when we undo the transitions they have triggered. Taking this into account, and assuming that the signals a and e are external signals, undoing the other events is unlikely to make much of a difference. Some events may occur in different orders, but we will most likely get new events corresponding to all the undone events. Only when *dbg(a)* or *dbg(e)* is undone, we make sure that undone transitions will not occur again once we resume execution of

the system. Using this way of undoing events, then, we have the choice of undoing *dbg(a)*, *dbg(e)*, both, or neither. While if we undo events in the order they occurred (as we currently do in our debugger), we would only have the choice of undoing *dbg(e)*, both *dbg(e)* and *dbg(a)*, or neither.

There are two things it could be useful to let the user of the debugger do with this. The first is to pick one event from a set of events that can be undone. The second is to pick any event that should be undone and then have the debugger find all events that must be undone before that one. This would give the user more flexibility when moving in time; in particular it would make it it easier to get one state machine back to a desired state while avoiding putting other state machines in undesired states. For example, we can say the *L3* opens a file during *dbg(e)*, and that our debugger does not detect the file being opened and cannot "undo" the opening of the file (the type of situation we discussed in *Event Effects* under section 8.2.2). Undoing *dbg(e)* while leaving the file open may be undesired. If the user wants to undo *dbg(a)* this would be possible by undoing the events in the order: *dbg(g), dbg(f), dbg(d), dbg(c), dbg(b), dbg(a)*. Which would avoid undoing *dbg(e)*.

If we assume that we have made the changes to the event model that we discussed in the previous section, finding events that can be undone is simple. For every state machine in the system we get its *prevEvent*, and if the event is at the end of a signal chain, given the chosen point in time, we can undo it. If it is not, we cannot undo it. To find out we must check every *SentSignal* in the event's *sent*. If one of the *SentSignal* instances has a *triggered* event, and that event is part of the debug controller's *past* (meaning that we have not already undone it), the event cannot be undone.

To find events that must be undone before a chosen event, we can:

1. Start with the event chosen by the user.
2. Find all the events that this event caused (*sent.triggered*), and the next event for the same state machine.
3. Add the events we found to an "undo" pool.
4. Repeat step 2–4 for all the events we found that were not already in the "undo" pool.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

The debugger works. We have found that there are some situations where reverting the state of the system under debugging is challenging (section 8.2.2). In practice, these have not proved to be big issues, though they may be for other state machine systems. However, these situations are non-issues if we're only *inspecting* the state of the system at different points in time.

The questionnaire results (section 5.3) indicate that the debugger has been useful to students. The back-in-time functionality was used by about half the students, while the trace of events was used by almost all the students. Improvements can be made to make the debugger easier to understand and use.

Judging by our test results (section 6.3), we can keep a couple of hundred thousand events in memory while still leaving most of the available memory free to use by the program under debugging. The exact amount of space used per event depends on the properties of the state machines in the system. We believe that this is good enough to be useful for a large number of different state machine systems (that the point in time for the cause of a bug will often be less than two hundred thousand transitions away from some symptom of that bug). We have also suggested a method for discarding less relevant events first, instead of just discarding the older events first (section 8.4.2). This may prove useful for systems where we must rely on events to (eventually) be discarded.

Our debugger cause some overhead in time per event; the amount of overhead depends on the state machine system that is running. In our tests, about a thousand events can be created and stored per second. And we have seen that this number increases drastically if we remove our GUI. For systems that do not require thousands of transitions to be executed each second, our debugger can be used. By replacing our GUI with one that is not updated as the debugged system is executing, the debugger should

be viable for many more systems. Adding support for selective debugging (section 8.3.3) should make the debugger viable for some systems that do need to execute transitions more frequently than the debugger can handle. With selective debugging, the debugger can create events only for transitions in one part of the state machine systems.

9.2 Future work

We have discussed some possible improvements in the previous chapter. When developing the debugger further, we should implement some of these:

- Replace the GUI with one that does not get updated during execution of the state machine system.
- Where possible, replace the use of Java's serialization API with something else; the results in section 6.3 indicate that we can get a performance boost from using it less.
- Add support for performing queries, and for selecting sets of state machines (section 8.3.2).
- Add support for selective debugging (section 8.3.3).
- Incorporate signal chains into our event model and add support for discarding "irrelevant" events, and for reordering events when moving back in time (section 8.4).
- Add support for running more schedulers at the same time (both regular schedulers and debug schedulers).

Added features should be tested and evaluated. In addition it would be useful to test the debugger on more state machine systems. In particular on "real" (that is, at least not constructed for testing specific debugger functionality) state machine systems where the overhead caused by the debugger is more likely to be an issue. This could give us an indication of what kind of improvements are more likely to be useful. E.g. if the overhead in time is a problem while the overhead in space is not, then spending more time on comparing "before" and "after" states of events in order to save space (as discussed in section 8.3.1) is not very useful.

9.2.1 The Modelling Tool/Debugger Connection

We are capable of establishing a connection between the debugger and modelling tool, as well as mapping elements in the running state machine system to model elements (section 4.7). However, we have not used that for very much. Currently, the only functionality we use it for is to highlight states

and transitions in diagrams. This can be improved and done for more UML elements:

- Improve the routine for highlighting transitions so that paths of transitions from one state to the next are highlighted (and not just the transition out from the "before" state; we discussed this issue in *JFT Compared to an "Ideal" UML Compiler* under section 2.3.2).
- Highlight classes, signals and state machines in class diagrams.
- In composite structure diagrams, highlight parts (state machines and composite classes). In addition, ports and connections between ports can be highlighted for sent signals.

We would also like to use the modelling tools to make selections in the debugger. We can translate from model elements to runtime elements just as easily as we can translate from runtime elements to model elements. Selecting state machines by using the modelling tool would be particularly useful for selecting state machines when using selective debugging (section 8.3.3). We think that the following functionality would be useful:

- Find all state machines of the state machine type selected in the modelling tool.
- Find all state machines that are in the state selected in the modelling tool.
- Find events for transitions triggered by a the signal type selected in the modelling tool. Or events for transitions that sent signals of that type.

As we saw in section 8.4.1, we can translate from our debugger events to sequence diagram events. It could be useful to generate traces of such events and check them against sequence diagrams in the modelling tool. We could check if a trace matched the weak sequencing of an interaction.

It would also be possible to move the GUI functionality to the Eclipse plug-in, so that using the debugger would be more like using a part of the modelling tool. The more we can make use of the modelling tool used, and the diagrams that the modellers of the system have made, the easier it should be for those modellers to understand what the debugger is talking about.

Bibliography

[1] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann, October 2005.

[2] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. *SIGPLAN Not.*, 40(6):15–26, 2005.

[3] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *SIGPLAN Not.*, 42(10):535–552, 2007.

[4] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), February 2006.

[5] OMG. OMG model driven architecture, `http://www.omg.org/mda/`, retrieved on 2009-04-18.

[6] OMG. MDA guide version 1.0.1, `http://www.omg.org/docs/omg/03-06-01.pdf`, 2003, retrieved on 2009-04-18).

[7] OMG. OMG's metaobject facility, `http://www.omg.org/mof/`, retrieved on 2009-04-18.

[8] OMG. Introduction to OMG's unified modeling language (UML), `http://www.omg.org/gettingstarted/what_is_uml.htm`, retrieved on 2009-04-18.

[9] OMG. UML 2, `http://www.uml.org/#UML2.0`, retrieved on 2009-04-21.

[10] Øystein Haugen, Birger Møller-Pedersen, and Thomas Weigert. Structural modeling with uml 2.0: classes, interactions and state machines. pages 53–76, 2003.

[11] The Eclipse Foundation. Eclipse, `http://www.eclipse.org/`, retrieved on 2009-04-18.

[12] The Eclipse Foundation. Eclipse modeling framework project (EMF), `http://www.eclipse.org/modeling/emf/`, retrieved on 2009-04-19.

[13] The Eclipse Foundation. Model development tools (MDT), `http://www.eclipse.org/modeling/mdt/`, retrieved on 2009-04-19.

[14] Papyrus. Papyrus UML, `http://www.papyrusuml.org`, retrieved on 2009-04-18.

[15] The Eclipse Foundation. Model to text (M2T), `http://www.eclipse.org/modeling/m2t/`, retrieved on 2009-04-21.

[16] M. Auer, T. Tschurtschenthaler, and S. Biffl. A flyweight uml modelling tool for software development in heterogeneous environments. *EUROMICRO Conference*, 0:267, 2003.

[17] Øystein Haugen and Birger Møller-Pedersen. B.: Javaframe: Framework for Java enabled modelling. In *In: Proc. Ericsson Conference on Software Engineering*, 2000.

[18] Bjørn Brændshøi. Consistency checking UML interactions and state machines. Master's thesis, University of Oslo, 2008.

[19] Henry Lieberman and Christopher Fry. *ZStep95: A reversible, animated source code stepper.*, pages 277–292. 1998.

[20] Bil Lewis. Omniscient debugging, `http://www.lambdacs.com/debugger/debugger.html`, retrieved on 2009-02-11.

[21] Bil Lewis. Debugging backwards in time. *Proceedings of AADEBUG '03*, 2003.

[22] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Design and implementation of a backward-in-time debugger. In Robert Hirschfeld, Andreas Polze, and Ryszard Kowalczyk, editors, *NODe/GSEM*, volume 88 of *LNI*, pages 17–32. GI, 2006.

[23] Mireille Ducassé. Opium: An extendable trace analyser for prolog. In *Special issue on Synthesis, Transformation and Analysis of Logic Programs*, 1999.

[24] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 592–615, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] A. Lienhard, O. Greevy, and O. Nierstrasz. Tracking objects to detect feature dependencies. pages 59–68, June 2007.

[26] Adrian Lienhard, Stéphane Ducasse, and Tudor Gîrba. Taking an object-centric view on dynamic information with object flow analysis. *Journal of Computer Languages, Systems and Structures*, 35(1):63–79, 2009.

[27] Stuart I. Feldman and Channing B. Brown. IGOR: a system for program debugging via reversible execution. *SIGPLAN Not.*, 24(1):112–123, 1989.

[28] Bob Boothe. Efficient algorithms for bidirectional debugging. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 299–310, New York, NY, USA, 2000. ACM.

[29] Bowen Alpern, Ton Ngo, Jong-Deok Choi, and Manu Sridharan. DejaVu: deterministic Java replay debugger for Jalape no Java virtual machine. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 165–166, New York, NY, USA, 2000. ACM.

[30] Shigeru Chiba. Load-time structural reflection in java. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 313–336, London, UK, 2000. Springer-Verlag.

[31] Darren Willis, David J. Pearce, and James Noble. Efficient object querying for java. In *ECOOP*, pages 28–49, 2006.

[32] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. STAIRS towards formal design with sequence diagrams. *Software and System Modeling*, 4(4):355–357, 2005.

[33] Ragnhild Kobro Runde, Øystein Haugen, and Ketil Stølen. The pragmatics of STAIRS. In *FMCO*, pages 88–114, 2005.

[34] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

Appendix A

Included CD

Because modifications must be made to Papyrus in order to make JavaFrame Transformation work, Papyrus is included on the CD. The **Papyrus-1.11.0-JFT** folder contains Papyrus, with JavaFrame Transformation and the latest version of JFDebug. Different ICU systems (examples used in the INF5150 course; most of the screenshots of JFDebug we have used have been taken while running ICU5) and the test systems from chapter 6 are included. This folder is all that is needed to test the debugger and plug-in.

The **Tests** folder contains everything needed to run the tests from chapter 6: the different JFDebug and JavaFrame versions used to run the tests, and Java class files for all the test systems. On a Windows machine, running **test1.bat** (requires Python) and **test2.bat** will run tests and generate the **instr*.txt** and **result*.txt** files. The text files are tab-formatted (and can be pasted into Microsoft Office Excel or OpenOffice Calc). The text files already in the folder are the results used in chapter 6. The scripts only run the processor usage tests; memory usage tests must be run "manually" (i.e. run each test system with the MAXEVENTS environment variable set to less than 0 and with different **-Xmx***n* settings). The versions of JFDebug included in this folder are the ones that were used to run the tests; some changes have been made JFDebug after the testing (e.g. the test versions cannot export sequence diagrams).

exeuctingICU0_guide.pdf and **inf5150_tool_guide_v1104.pdf** are tool guides for the INF5150 course, by Rayner Vintervoll[1]. For information on how to use Papyrus and JavaFrame Transformation, these can be consulted.

The **UMLet 9.1** folder contains the stand-alone version of UMLet 9.1. It can be used to view sequence diagrams exported with JFDebug.

fakepats.jar can be used to run *Fake PATS Central*. To run the ICU systems, Fake PATS must be started first (`java -jar fakepats.jar`).

[1] Also available at `http://www.uio.no/studier/emner/matnat/ifi/INF5150/h08/undervisningsmateriale/papyrus-ifi-uml/docs/`

JFDebug.jar is the latest version of JFDebug.

JFDebug_course.jar is the version of JFDebug that was used in the *INF5150 – Unassailable IT-systems* (chapter 5).

no.jonaw.jfdebug_1.0.0.jar is the JFDebug Eclipse plug-in.

Sources are included in all JFDebug jar files (i.e. the different JFDebug versions and the Eclipse plug-in).

Appendix B

JFDebug User Instructions

To use JFDebug instead of regular JavaFrame, a JavaFrame system must be started with **JFDebug.jar**, instead of **JavaFrame*.jar**, in the classpath. The projects in the Papyrus versions on the included CD are configured to use JFDebug, and new JavaFrame projects should use JFDebug by default.

The "ICUx" project in Papyrus already contains code generated from the "ICU5" system, and a launch configuration is made for it. So in order to run that system, it is just to open the drop down menu for "Run" and select "ICUsystemMain" (figure B.1). Fake PATS should be started before any of the ICU systems.

The state machine system will start in debug mode. Pushing the "Stop debugging" button turns debugging off (and turns "Stop debugging" into "Start debugging"). And pushing "Start debugging" when debugging is turned off turns debugging on again.

B.1 Inspecting the Debugged System

The main window (figure B.2) starts with the void tab selected. The "void" tab is the only one that does not get updated during execution of the debugged system, and it is only there because the system executes faster when

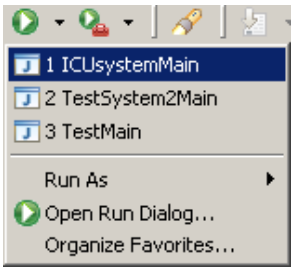


Figure B.1: The Eclipse "Run" drop down menu

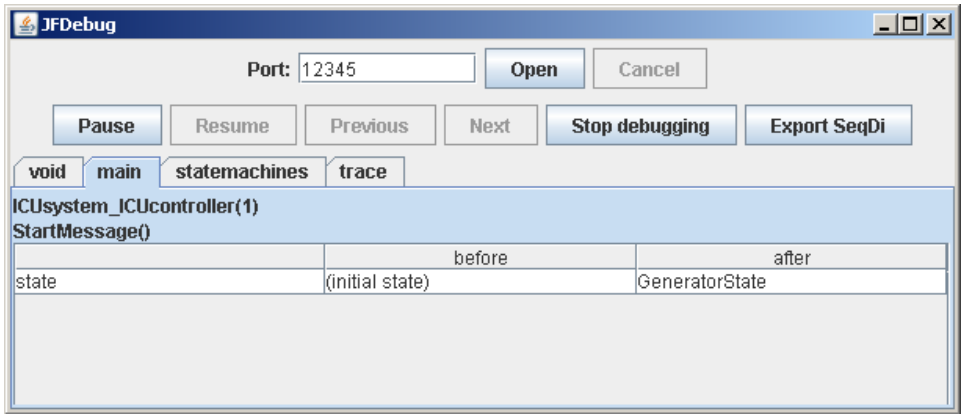


Figure B.2: JFDebug's main window

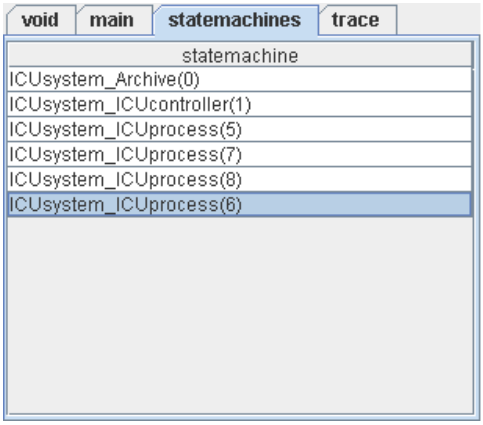


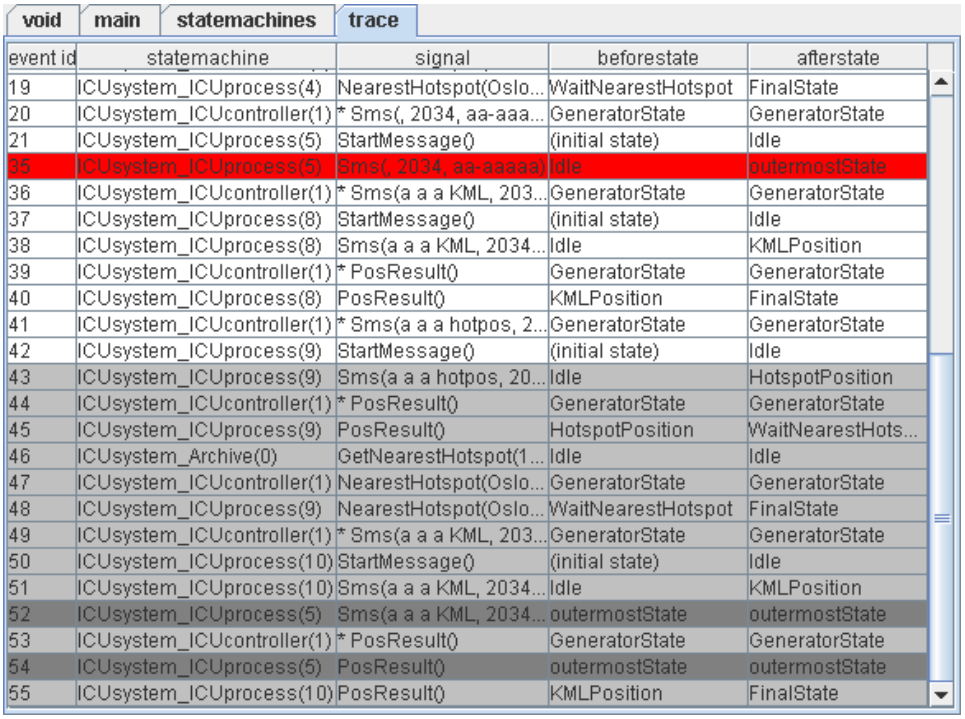
Figure B.3: The state machines tab

the GUI is not redrawn between the transitions (having void selected rather than trace or main makes a noticeable difference in the test systems).

The "main" tab shows the most recently occurred event for the debugger's point in time: when the debugged system is executing, it gets updated as new transitions execute, and when the system is paused it gets updated as we move in time.

The "statemachines" (figure B.3) tab shows a list of all the state machines in the debugged system at the debugger's point in time. Double-clicking a state machine opens a state machine window. Each state machine window shows the most recently occurred event for the state machine it is for, given the debugger's point in time. Any amount of state machine windows can be opened. Several state machine windows can be seen in figure B.5.

The "trace" (figure B.4) tab shows a trace of all occurred transitions/events. Each row represents one event. When moving in time, the events that occurred later than the debugger's point in time are coloured grey. I.e. grey



event id	statemachine	signal	beforestate	afterstate
19	ICUsystem_ICUprocess(4)	NearestHotspot(Oslo...	WaitNearestHotspot	FinalState
20	ICUsystem_ICUcontroller(1)	* Sms(, 2034, aa-aaa...	GeneratorState	GeneratorState
21	ICUsystem_ICUprocess(5)	StartMessage()	(initial state)	Idle
35	ICUsystem_ICUprocess(5)	Sms(, 2034, aa-aaaaa)	Idle	outermostState
36	ICUsystem_ICUcontroller(1)	* Sms(a a a KML, 203...	GeneratorState	GeneratorState
37	ICUsystem_ICUprocess(8)	StartMessage()	(initial state)	Idle
38	ICUsystem_ICUprocess(8)	Sms(a a a KML, 2034...	Idle	KMLPosition
39	ICUsystem_ICUcontroller(1)	* PosResult()	GeneratorState	GeneratorState
40	ICUsystem_ICUprocess(8)	PosResult()	KMLPosition	FinalState
41	ICUsystem_ICUcontroller(1)	* Sms(a a a hotpos, 2...	GeneratorState	GeneratorState
42	ICUsystem_ICUprocess(9)	StartMessage()	(initial state)	Idle
43	ICUsystem_ICUprocess(9)	Sms(a a a hotpos, 20...	Idle	HotspotPosition
44	ICUsystem_ICUcontroller(1)	* PosResult()	GeneratorState	GeneratorState
45	ICUsystem_ICUprocess(9)	PosResult()	HotspotPosition	WaitNearestHots...
46	ICUsystem_Archive(0)	GetNearestHotspot(1...	Idle	Idle
47	ICUsystem_ICUcontroller(1)	NearestHotspot(Oslo...	GeneratorState	GeneratorState
48	ICUsystem_ICUprocess(9)	NearestHotspot(Oslo...	WaitNearestHotspot	FinalState
49	ICUsystem_ICUcontroller(1)	* Sms(a a a KML, 203...	GeneratorState	GeneratorState
50	ICUsystem_ICUprocess(10)	StartMessage()	(initial state)	Idle
51	ICUsystem_ICUprocess(10)	Sms(a a a KML, 2034...	Idle	KMLPosition
52	ICUsystem_ICUprocess(5)	Sms(a a a KML, 2034...	outermostState	outermostState
53	ICUsystem_ICUcontroller(1)	* PosResult()	GeneratorState	GeneratorState
54	ICUsystem_ICUprocess(5)	PosResult()	outermostState	outermostState
55	ICUsystem_ICUprocess(10)	PosResult()	KMLPosition	FinalState

Figure B.4: The JFDebug trace tab

events are in the "future" and white events are in the "past". The "signal" column shows the names of the signals that triggered the different transitions; external signals are marked with asterisks. Events for transitions where exceptions occurred are coloured red (or dark grey if they are in the future). Events can be double-clicked to open event windows.

Information about state machines and events are shown in "event views". The event view for a state machine shows the most recently occurred event for that state machine. The event view shows:

- Which state machine the event is for.
- Which signal triggered the transition the event is for.
- The state and property values from before and after the event/transitions.

Event views can be seen in the main tab in figure B.2 and in the state machine windows in figure B.5.

Sequence diagrams can be exported by pressing the "Export SeqDi" button. The sequence diagrams are saved as .uxf files and can be opened with UMLet 9.1.

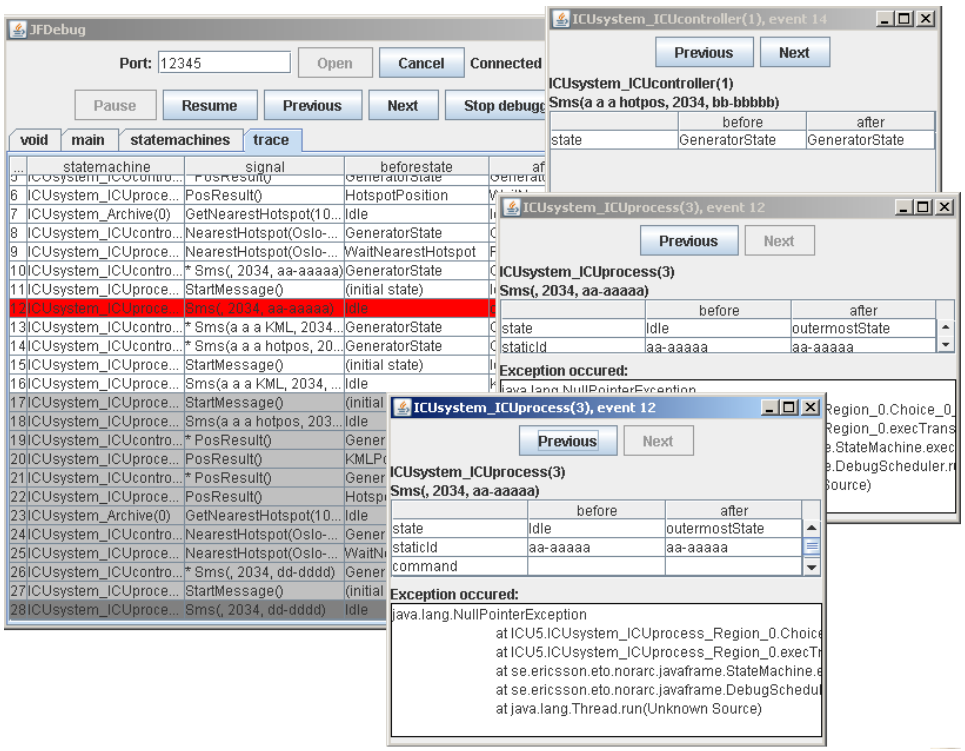


Figure B.5: The GUI in use. Several state machine windows are opened and the main window is showing the event trace

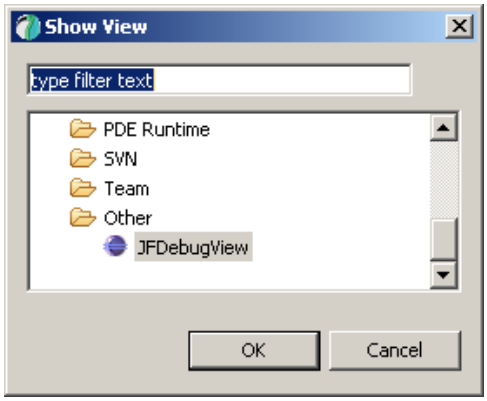


Figure B.6: The JFDebug view is in the "Other" category

B.2 Moving in Time

The debugged system can be paused by pressing the "Pause" button. Once the system is paused, it is possible to move in time. There are three ways to navigate in time:

- Pressing the "Previous" and "Next" buttons in the main window. This steps one transition back or forward in time.
- Pressing the "Previous" and "Next" buttons in state machine windows. This moves back in time until the most recently occurred event for that state machine changes.
- Selecting an event in the trace tab, then right clicking it and selecting "Go to". This moves to the point in time where the selected event was the most recently occurred event.

Execution of the debugged system can be resumed again by pushing the "Resume" button. This resumes execution from the point in time we have moved to. If the state of the debugged system should not be reverted to what it was at an earlier point in time, we must move forward in time, as far as possible, before resuming execution (i.e. until the "Next" button in the main window cannot be pushed, and all the events in the trace tab are coloured white (or red)). If the state of the system is reverted to that of an earlier point in time, any state machine windows for state machines that did not exist at that point in time are removed.

B.3 Using the Eclipse plug-in

To use the JFDebug Eclipse plug-in, the "JFDebugView" must be opened. Open the "Window" menu in Papyrus/Eclipse, then select "Show View" and

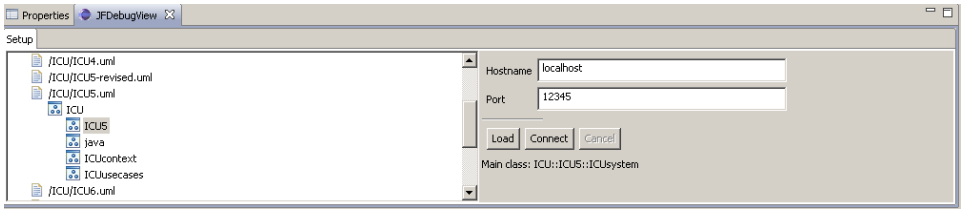


Figure B.7: *The JFDebug view*

"Other...". Then select "JFDebugView" from the "Other" category (figure B.6).

JavaFrame Transformation is used on UML packages. In the JFDebug view, the UML package that was used for the JFT transformation must be selected. The JFDebug view shows a tree list of UML files found in the Eclipse workspace and their packages. Usually, a UML file contains one model (e.g. *ICU* in figure B.7) that contains the package that was used for the transformation (e.g. *ICU5* in the figure).

When the correct model is selected, it can be loaded by pressing "Load" in the JFDebug view. To make sure the package could have been used in a JFT transformation, it will look for a main class (i.e. one with a Composite stereotype from the JavaFrame UML profile, and main set to true). If it finds on, the "Connect" button becomes available.

Now we can connect the debugger and plug-in to each other. We must run the code transformed from the package that was selected in the plug-in, and use the debugger. The same port number must be selected in the debugger (both are set to 12345 by default) and the plug-in, and "Open" must be pushed in the debugger window before "Connect" is pushed in the plug-in. Unless some error occurs, both the debugger and the plug-in should now say "Connected". As we move in time, and as the debugged system is executing, the most recently occurred event will now be highlighted in the Papyrus diagrams for the model that was selected. Like in figure B.8.

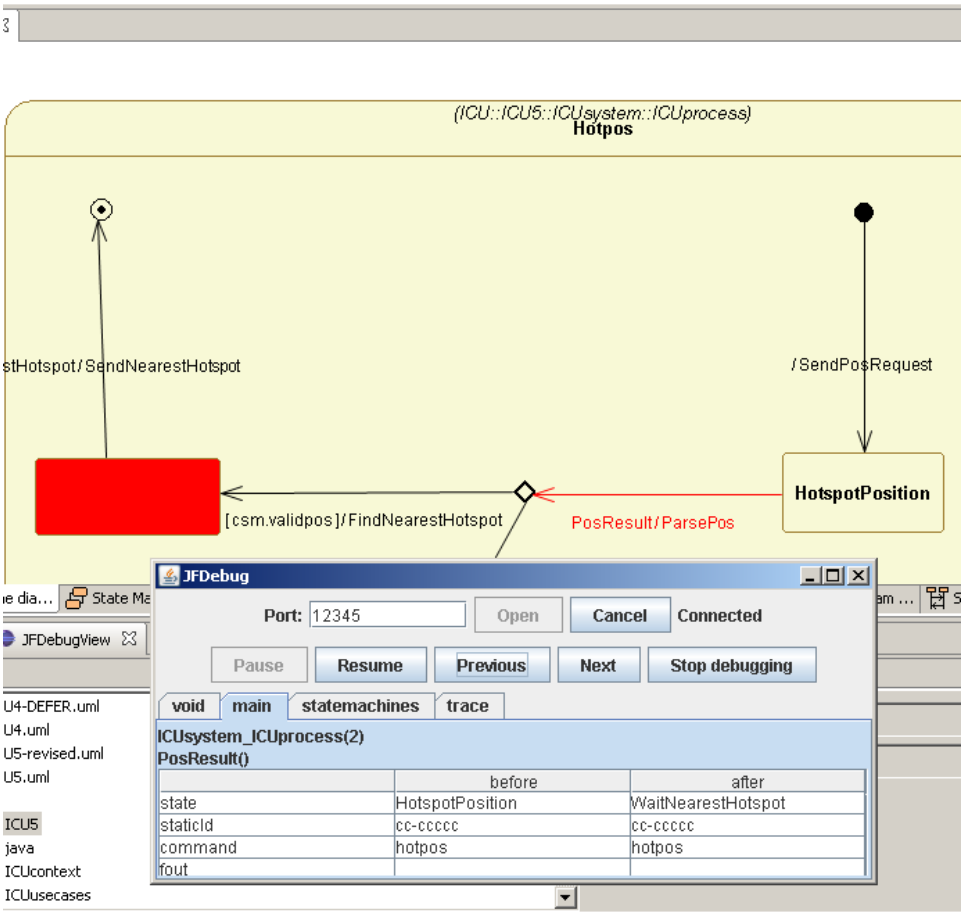


Figure B.8: The plug-in highlights the debugger's most recently occurred event